PROCESSOS DE DESENVOLVIMENTO DE SOFTWARE

AUTOR
SAULO AMUI

PROCESSOS DE DESENVOLVIMENTO DE *SOFTWARE*

AUTOR

SAULO FRANÇA AMUI

1ª EDIÇÃO

SESES

RIO DE JANEIRO 2015

Conselho editorial REGIANE BURGER; ROBERTO PAES; GLADIS LINHARES; KAREN BORTOLOTI; HELCIMARA AFONSO DE SOUZA

Autor do original SAULO FRANÇA AMUI

Projeto editorial ROBERTO PAES

Coordenação de produção GLADIS LINHARES

Coordenação de produção EaD KAREN FERNANDA BORTOLOTI

Projeto gráfico PAULO VITOR BASTOS

Diagramação BFS MEDIA

Revisão linguística BFS MEDIA

Imagem de capa SEMISATCH | DREAMSTIME.COM

Dados Internacionais de Catalogação na Publicação (CIP)

A529p Amui, Saulo
Processos de desenvolvimento de software / Saulo Amui.
Rio de Janeiro : SESES, 2015.
176 p. : il.

ISBN: 978-85-5548-040-9

1. Processos de software. 2. Engenharia de software.
3. Requisitos. 4. Métodos de desenvolvimento. I. SESES. II. Estácio.

CDD 005.1

Diretoria de Ensino — Fábrica de Conhecimento
Rua do Bispo, 83, bloco F, Campus João Uchôa
Rio Comprido — Rio de Janeiro — RJ — CEP 20261-063

Sumário

4. Modelos de Ciclo de Vida de *Software* 97

5. Processo Unificado (UP) 135

Prefácio

Prezados(as) alunos(as),

O conhecimento dos Processos de Desenvolvimento de *Software* é essencial para o desenvolvimento e construção de um produto de *software* de qualidade. Veremos que a qualidade de *software* está associada a seus processos e às boas práticas de desenvolvimento.

Desde o seu início, a área de desenvolvimento de *software*s encontra grandes desafios para obter sucesso nos projetos propostos destas, já que trata-se de uma atividade complexa e cheia de fatores que influenciam as atividades de desenvolvimento e, consequentemente, o êxito do projeto de *software*. Desta forma, o estudo na área de Engenharia de *Software* trouxe uma série de métodos, processos, modelos e normas para que os profissionais desta área pudessem ter o auxílio necessário para saber lidar com estes desafios, presentes em praticamente todas as etapas do processo de desenvolvimento.

Cada fase ou etapa do processo de desenvolvimento de *software* tem uma infinidade de assuntos importantes e que foram estruturados nas diferentes metodologias existentes. A documentação de *software* também é algo que está presente por todas estas etapas.

Ao longo destes capítulos, você poderá conhecer e aprender muitas coisas importantes sobre os aspectos que envolvem o desenvolvimento de um *software* e verá que, mesmo sendo uma atividade complexa, do ponto de vista do gerenciamento do projeto, há diferentes meios e maneiras de realizá-la, com adaptações corretas e conhecendo bem os principais pontos que cada modelo, método e processo propõem para as diferentes situações.

Por fim, vale ressaltar que, embora estejamos abordando vários pontos importantes ao longo do livro, há uma imensidão de conteúdos, materiais e referências bibliográficas de praticamente todos os assuntos aqui tratados, valendo a pena pesquisar e estudar ainda mais os pontos de maior interesse e necessidade em sua vida profissional.

Aproveite a leitura e os estudos para tentar transportar os conceitos aprendidos em sua vida prática, ou observá-los em uso em empresas e nos ambientes adequados.

Bons estudos!

1

Conceitos Gerais e Atividades de Processo de Desenvolvimento de *Software*

Neste capítulo, você aprenderá sobre as atividades do processo de desenvolvimento de *software*, que fornece metodologias para as práticas da Engenharia de *software*.

O processo de *software* é visto por uma sequência de atividades que produzem uma variedade de documentos, resultando em um programa satisfatório e executável.

A partir do estudo destes assuntos, você poderá compreender melhor o que é um processo de *software*, saber quais são as principais atividades genéricas que estão presentes na maioria dos processos.

O desenvolvimento de *software* envolve uma série de atividades sistematizadas que podem ser estruturadas em fases ou etapas. Veremos, ainda, a importância dos requisitos de *software* que norteiam estas atividades e é a base fundamental para a compreensão do que será desenvolvido.

Por ser uma atividade complexa, o desenvolvimento de *software* também possui alguns problemas comuns e, por conta destas situações, também é importante ressaltar a análise de requisitos, que envolve algumas etapas para garantir a qualidade dos processos e diminuir os riscos e problemas. A especificação clara e descritiva dos requisitos irá determinar a modelagem ou desenho do *software*, para então ser codificado (implementado) e depois testado.

De acordo com (LEITE, 2000), a elaboração do processo de desenvolvimento de *software* é determinante para detalhar quem irá fazer o que, quando e como dentro das atividades que envolvem este desenvolvimento. Assim, um processo pode ser considerado como sendo uma instância de um método com suas técnicas e ferramentas associadas, que são elaboradas durante a fase de planejamento e que possuem suas atividades distribuídas a cada membro da equipe de desenvolvimento, com datas de entregas definidas e outros pontos de controle para avaliação e controle de seu andamento.

OBJETIVOS

- Compreender os principais conceitos que envolvem as atividades de processo de desenvolvimento de software;
- Conhecer os problemas mais comuns no processo de desenvolvimento de software;
- Ter uma visão da análise econômica e de requisitos de software;
- Aprender sobre a especificação do software e desenho ou arquitetura do sistema de software;
- Conhecer as atividades que envolvem a codificação ou implementação de software;
- Conhecer os principais conceitos e atividades relacionadas aos testes.

1.1 Introdução

Desenvolver *softwares* é uma atividade complexa e não poderia ser realizada se não houvesse um conjunto de atividades, sequências, devidamente organizadas e estruturadas em passos ou etapas bem definidas. Este conjunto de atividades é considerado um processo e, quando o aplicamos em um contexto de *software*, estamos lidando com os processos de desenvolvimento de *software*, auxiliados pelas áreas de conhecimento da Engenharia de *software*.

Quando o homem, ao longo de sua história, precisou construir ou elaborar grandes construções, a engenharia contribuiu para a melhoria dos processos, encontrando os melhores recursos e otimizando as atividades por meios de métodos que foram amadurecendo cada vez mais, assim como podemos notar na engenharia civil, na construção de grandes prédios ou grandes obras, onde estas construções são divididas em etapas muito claras e com as suas peculiaridades e especificidades. Para a construção de *softwares*, estas práticas da engenharia também estão presentes quando notamos os processos utilizados e as técnicas aplicadas nas diferentes fases, envolvendo desde a sua concepção, passando por modelagens do que será construído, pelo desenho do *software*, pela discussão e análise ainda no papel, para só então, depois, partir para uma fase específica de codificação ou de implementação e, posteriormente, para a etapa de testes, e assim serem entregues aos clientes ou usuários finais.

Veremos, ao longo dos próximos tópicos, maiores detalhes das principais etapas que envolvem estes processos de desenvolvimento de *software*, conhecendo um pouco mais de detalhes sobre estes processos e das situações que os envolvem.

1.2 O que é? Para que serve?

Chamaremos de processo de *software* o conjunto de atividades, métodos, práticas e transformações usados para desenvolver e manter produtos de *software*. Lembramos que o conceito de produto de *software* inclui os artefatos associados, como documentos e modelos. Os processos de *software* são processos de negócio das organizações desenvolvedoras e mantenedoras de *software*.

Nas organizações imaturas, os processos geralmente são informais. O problema é que essa informalidade é individual, ou seja, a organização não

entende isso como oficial e tampouco os conhece. Estes processos podem ser parcialmente transferidos para outras pessoas, por transmissão oral e por imitação, e isso é um grande problema.

Em contrapartida, quando existe um processo definido e tem documentação detalhada sobre todos os seus aspectos importantes, por exemplo, o que é feito, quando, por quem, as coisas que usa e as coisas que produz, a organização como um todo cresce.

Existem muitas formas de representação de processos definidos. Exemplos de possíveis formas são:

- por uma sequência de passos descritos em linguagem natural;
- por tabelas nas quais cada linha corresponde a um passo do processo;
- por representações gráficas, como os fluxogramas e notações equivalentes, entre elas a UML.

Sem dúvida, quando os processos são bem definidos, a maturidade das organizações produtoras de *software* é incrementada, pois permitem que a organização tenha uma forma de trabalho padronizada e que pode ser repetida. Desta forma, a capacitação das pessoas é facilitada e torna o funcionamento da organização menos dependente de determinados indivíduos.

Entretanto, os processos definidos não serão sinônimos de plena realização. Processos rigorosamente definidos, mas não alinhados com os objetivos da organização, tornam-seimpedimentos burocráticos e não fatores de produção.

Vimos que para tornar uma organização mais madura e capacitada é realmente preciso melhorar a qualidade dos seus processos. Os processos não melhoram simplesmente por estarem de acordo com um padrão externo, e sim pela medida de quanto eles contribuem para que os produtos sejam entregues aos clientes e usuários:

- com melhor qualidade;
- com menor custo;
- e num menor prazo.

A ISO/IEC, 2008, também define processo como sendo um conjunto de atividades inter-relacionadas que podem transformar entradas em saídas. Estas entradas são também conhecidas por requisitos do cliente ou até mesmo por

produtos intermediários que são produzidos pelos processos ao longo do desenvolvimento de *software*. As saídas também podem ser estes produtos intermediários, que serão utilizados por outros processos ou pelo produto de *software* final (COSTA, 2012, p. 9).

No âmbito de desenvolvimento de *software*, processo e ciclo de vida de *software* estão diretamente relacionados. Assim, pode-se considerar que processo de *software* está associado a um conjunto coerente de políticas, estruturas organizacionais, procedimentos, artefatos e tecnologias que são necessários para conceber, desenvolver, entregar e manter um produto de *software* (FUGGETTA, 2000); e ainda, de acordo com PAULK et al., 1993, a produtos associados, como código-fonte, manual, plano do projeto, casos de uso, casos de testes, etc. (COSTA, 2012, p. 9).

Sommerville (2003) ainda define processo de *software* como "um conjunto de atividades e resultados associados que geram um produto de *software*". A maioria destas atividades são executadas por engenheiros de *software*, sendo que existem quatro atividades de processo fundamentais e comuns a todos os processos de *software*. Ainda de acordo com Leite (2000), essas atividades são:

1. ESPECIFICAÇÃO DO *SOFTWARE*	A funcionalidade do *software* e as restrições em sua operação devem ser definidas.
2. DESENVOLVIMENTO DO *SOFTWARE*	O *software* deve ser produzido de modo que atenda a suas especificações.
3. VALIDAÇÃO DO *SOFTWARE*	O *software* tem de ser validado para garantir que ele faça o que o cliente deseja.
4. EVOLUÇÃO DO *SOFTWARE*	O *software* deve evoluir para atender às necessidades mutáveis do cliente.

SOMMERVILLE, 2003

Essas atividades são organizadas em diferentes processos, de diversas maneiras e níveis de detalhamento e possuem prazos variados e resultados que também podem variar. Para se desenvolver um produto, diferentes organizações podem adotar processos distintos, mesmo que o produto final seja semelhante ou igual. Mas é importante perceber e considerar que determinados processos são melhores para uma situação do que outros, e por isso alguns processos devem ser adequados para alguns tipos de aplicação. Caso um processo inadequado seja utilizado, isto poderá comprometer e impactar na qualidade do produto final que está sendo desenvolvido (SOMMERVILLE, 2006).

Afinal, qual a definição de processo? Wazlawick, 2013, p. 30, traz em seu livro que, segundo Sommerville (2003), "processo é um conjunto de atividades e resultados associados que geram um produto de *software*".

Conforme Wazlawick, 2013, p. 30, comenta, essa definição direta e objetiva também apresenta uma simplicidade que pode limitar a sua definição, pois processos não são apenas conjunto de atividades, mas sim atividades estruturadas dentro de um contexto, considerando outros aspectos envolvidos, como pessoas, restrições, recursos, padrões a serem seguidos, limitações e outros fatores envolvidos em um processo.

Há várias definições e apresentações do conceito sobre processo de engenharia de *software* e a Wikipédia apresenta uma definição interessante e completa:

> Um processo de engenharia de *software* é formado por um conjunto de passos de processo parcialmente ordenados, relacionados com artefatos, pessoas, recursos, estruturas organizacionais e restrições, tendo como objetivo produzir e manter os produtos de *software* finais requeridos" (WAZLAWICK, 2013, p. 30).

Desta forma, podemos entender processo como um conjunto de atividades que são interdependentes, com responsáveis e com entradas e saídas definidas. Ou, ainda, como um "conjunto de regras que definem como um projeto deve ser executado" (WAZLAWICK, 2013, p. 31).

O Processo de *Software*

O autor Silva (2006) cita uma consideração importante levantada por Fuggeta (2000) e outra observação realizada por Rocha et al. (2001), nos respectivos trechos a seguir:

"Uma importante contribuição da área de pesquisa de processo de *software* tem sido a conscientização de que o desenvolvimento de *software* é um processo complexo. Pesquisadores e profissionais têm percebido que desenvolver *software* não está circunscrito somente à criação de linguagens de programação e ferramentas efetivas. O desenvolvimento de *software* é um processo coletivo, complexo e criativo. Sendo assim, a qualidade de um produto de *software* depende fortemente das pessoas, da organização e de procedimentos utilizados para criá-lo e disponibilizá-lo."

"Dada a complexidade envolvida na definição de processos de *software*, não é uma boa estratégia definir cada processo de projeto a partir do zero. Assim, apesar de cada projeto ter suas características próprias, é possível estabelecer conjuntos de elementos que devem estar presentes em todos os processos de uma organização. Esses elementos em comum possibilitam a formação dos processos-padrão da organização, que, por sua vez, podem ser especializados para determinadas classes de projetos dessa organização. Processos-padrão e especializados podem, então, ser instanciados em processos de projeto, em uma abordagem de definição de processos em níveis"

Nas empresas, um processo é considerado como sendo um conjunto de regras específicas que seus colaboradores devem adotar para serem seguidas quando lidam com projetos. Desta forma, uma vez que o processo é definido, os gerentes ou responsáveis definem as atividades relativas, bem como os prazos e as atribuições das atividades que deverão ser desenvolvidas ao longo do projeto.

Quando os processos possuem um conjunto de regras de maneira mais abstrata e estes são especificados, o consideramos como um **modelo de processo** (WAZLAWICK, 2013, p.31).

O modelo de processo adotado em um projeto também é conhecido e chamado de ciclo de vida. Desta forma, quando uma empresa seleciona e define seus processos, ela deve buscar um modelo e adaptar a filosofia e as melhores práticas para desenvolver e elaborar o seu próprio processo que serão utilizados e seguidos na empresa (WAZLAWICK, 2013, p. 31).

Atualmente, existem vários modelos de processos de *software*, mas neste momento iremos apenas citar 3 categorias destes modelos para exemplificar: cascata, desenvolvimento evolucionário e baseado em componentes.

Wazlawick, 2013, p. 31, apresenta algumas vantagens em definir o desenvolvimento de *software* como um processo, como pode-se ver nesta pequena relação a seguir:

> **a) O tempo de treinamento pode ser reduzido:** com processos bem definidos e documentados, é mais fácil encaixar novos indivíduos na equipe do que quando não se dispõe deles.
>
> **b) Produtos podem ser mais uniformizados:** a existência do processo não garante a uniformidade na qualidade dos produtos, mas certamente uma equipe com um processo bem definido tende a ser mais previsível do que a mesma equipe sem processo algum.
>
> **c) Possibilidade de capitalizar experiências:** pode-se imaginar que um processo de trabalho bem definido poderia tolher o uso da criatividade dos desenvolvedores. Contudo, isso não é verdade, a não ser que a empresa tenha processos engessadores, o que não é bom. Um processo bem gerenciado deve ter embutidos mecanismos para melhoria. Assim, se um desenvolvedor descobrir um meio de fazer as coisas de maneira melhor do que a que está descrita no processo, devem haver meios para incorporar essas alterações.

Após estas definições acerca de processos de *software*, Marcoratti (2014) sintetiza e considera que de maneira geral estes processos são um "conjunto de atividades, métodos, ferramentas e práticas que são utilizados para construir um produto de *software*". A partir desta definição, vale ressaltar que devemos considerar ainda o modelo de ciclo de vida utilizado, bem como as atividades que deverão ser realizadas, os recursos utilizados e todos os artefatos que serão necessários e desenvolvidos ao longo do processo.

Em seu artigo, Marcoratti (2014) cita as razões para a definição de um processo-padrão, como a redução de problemas que poderiam estar associados a treinamento, revisões e suporte a ferramentas, a aquisição de experiências ao longo do desenvolvimento de projetos, que poderiam ser incrementadas ao processo-padrão e proporcionar melhorias nestes processos definidos, e economia de tempo e esforço no momento de definir outros novos processos que seriam adequados aos respectivos projetos.

1.2.1 Fases de um processo de *Software*

Vamos conhecer um pouco mais das principais fases e atividades comuns nos modelos de desenvolvimento de *softwares*.

1. ESPECIFICAÇÃO DE REQUISITOS	Envolve uma análise dos requisitos, independentemente do modelo de processos adotado para se desenvolver um *software*. Será necessário conhecer quais são as necessidades deste *software*, quais são as restrições e conhecer bem o domínio da aplicação.
2. PROJETO DE SISTEMA	Nesta fase envolve o projeto propriamente dito, a partir de análises e tradução dos requisitos para que estes possam ser codificados na próxima fase e permitir uma capacidade de gerência.
3. PROGRAMAÇÃO (CODIFICAÇÃO)	Desenvolvimento do código-fonte em uma linguagem de programação, que irá representar o controle do sistema, a lógica e as operações de acordo com aquilo que fora descrito e levantado na fase de projetos a partir dos requisitos.
4. VERIFICAÇÃO E INTEGRAÇÃO (VALIDAÇÃO)	Nesta fase são realizados os testes, a verificação do sistema checando se o produto desenvolvido ou seus componentes estão alinhados com as necessidades propostas pelo cliente e a satisfação destes para com o *software* ou sistema desenvolvido, visando a qualidade do *software*. Nesta fase também acontece a integração, colocando o sistema em produção para qe seus usuários finais possam usá-lo..
5. MANUTENÇÃO E EVOLUÇÃO	Esta fase prevê as mudanças, ajustes, reparos, melhorias e qualquer alteração que os requisitos venham sofrer após o sistema ser colocado em produção. Esta fase também é conhecida como manutenção e estas atividades podem sofrer tanto mudanças estruturadas como não estruturadas.

É muito importante ressaltar neste ponto que não há um modelo rígido e uma sequência obrigatória que devem ser seguidos literalmente sem considerar a realidade do negócio, da organização e do contexto ao qual está inserido. Desta forma, não podemos apontar uma sequência obrigatória de fases e, em algumas situações, estas podem ocorrer até mesmo de modo paralelo. Vale ainda ressaltar que há modelos interativos, onde existe uma forte interação com o cliente, e estas fases também ocorrem de modo cíclico, evoluindo a cada volta deste ciclo.

1.2.2 Atividades do Processo de *Software*

Em cada fase de um processo de *software* são realizadas atividades específicas para que o propósito da respectiva fase seja alcançado, de modo que se obtenha o objetivo final, que é o produto de *software*.

Marcoratti (2014) realiza uma combinação de classificações feitas por diferentes autores e lista as principais atividades de cada fase:

1. **Especificação**

a) **Engenharia de Sistema**: estabelecimento de uma solução geral para o problema, envolvendo questões extra software.

b) **Análise de Requisitos**: levantamento das necessidades do software a ser implementado. A Análise tem como objetivo produzir uma especificação de requisitos, que convencionalmente é um documento.

c) **Especificação de Sistema**: descrição funcional do sistema. Pode incluir um plano de testes para verificar adequação.

2. **Projeto**

a) **Projeto Arquitetural**: onde é desenvolvido um modelo conceitual para o sistema, composto de módulos mais ou menos independentes.

b) **Projeto de Interface**: onde cada módulo tem sua interface de comunicação estudada e definida.

c) **Projeto Detalhado**: onde os módulos em si são definidos, e possivelmente traduzidos para pseudocódigo.

3. **Implementação**

a) **Codificação**: a implementação em si do sistema em uma linguagem de computador.

4. **Validação**

a) **Teste de Unidade e Módulo**: a realização de testes para verificar a presença de erros e comportamento adequado a nível das funções e módulos básicos do sistema.

b) **Integração**: a reunião dos diferentes módulos em um produto de software homogêneo e a verificação da interação entre estes quando operando em conjunto.

5. **Manutenção e Evolução**

a) Nesta fase, o software entra em um ciclo iterativo que abrange todas as fases anteriores.

Sendo assim, o produto final está associado às atividades de cada fase do processo de desenvolvimento, que por sua vez são especificadas, detalhadas e colocadas em uma ordem que auxilia e orienta os diferentes modelos de desenvolvimento de *softwares*.

CONEXÃO

Você pode encontrar recursos completos para padrões de processo no endereço eletrônico: http://ambysoft.com/processPatternsPage.html

No livro de Engenharia de *Software* do conhecido e respeitado autor Pressman (PRESSMAN, 2011, p. 58) há uma pergunta com uma resposta da Nasa. A pergunta: "Quais técnicas formais estão disponíveis para avaliar o processo de *Software*?". A resposta dada pela Nasa é: "As organizações de *Software* apresentaram falhas significativas quanto à habilidade em capitalizar as experiências adquiridas nos projetos finalizados".

1.3 Problemas mais comuns

Há uma significativa complexidade ao resolver problemas na área de engenharia de *software*, pois entender com exatidão e precisão a origem e até mesmo a natureza dos problemas não é uma tarefa fácil. Além do mais, o fato de definir com precisão o que o sistema irá fazer também é uma ação bem difícil. Os **requisitos de sistema** descrevem as funções de um sistema, suas limitações e restrições, enquanto que a **engenharia de requisitos** abrange as atividades

do processo de desenvolvimento, com descobertas, análises, documentações, verificações e validações. Os requisitos podem ser declarados de maneira abstrata, com uma linguagem de alto nível (da forma como pensamos e falamos), além destas definições funcionais de um sistema (SOMMERVILLE, 2006, p. 82).

Os diferentes níveis de descrição dos requisitos resultam em alguns dos problemas neste processo de engenharia de requisitos, pois existem aqueles que vêm dos usuários com descrições de alto nível, de forma abstrata, chamados **requisitos do usuário**, e existem os requisitos de sistema, que descrevem de maneira mais detalhada como uma função deverá se comportar no sistema. Além destes dois tipos, ainda pode haver um detalhamento mais especifico na fase de especificação do projeto, associando estes requisitos às atividades que deverão ser realizadas (SOMMERVILLE, 2006, p. 82). O ponto de atenção aqui, para evitar problemas, está no desafio de se obter uma descrição detalhada e precisa dos requisitos de usuário, já que estes são obtidos de uma maneira muito subjetiva e que, por este motivo, pode "esconder" desdobramentos de outros requisitos que resultariam de mais atividades no projeto que não estariam previstas inicialmente, comprometendo a qualidade do produto final e o sucesso no projeto, no que se refere ao escopo, custo e prazo.

Sommerville (2006, p. 82) define os requisitos do usuário, os requisitos de sistema e a especificação de projeto de *software* da seguinte maneira:

1. REQUISITOS DO USUÁRIO	São declarações, em linguagem natural e também em diagramas, sobre as funções que o sistema deve fornecer e as restrições sob as quais deve operar.
2. REQUISITOS DE SISTEMA	Estabelecem detalhadamente as funções e as restrições do sistema. O documento de requisitos de sistema, algumas vezes chamado de especificação funcional, deve ser preciso. Ele pode servir como um contrato entre o comprador do sistema e o desenvolvedor do *software*.
3. ESPECIFICAÇÃO DE PROJETO DE *SOFTWARE*	É uma descrição abstrata do projeto de *software* que é uma base para o projeto e a implementação mais detalhados. Essa especificação acrescenta mais detalhes à especificação de requisitos do sistema.

SOMMERVILLE, 2006, p. 82.

Zabeu et al. (2006) comenta que muitos problemas que envolvem as organizações estão além da falta de clareza dos requisitos ou da ausência de testes que que deveriam ser aplicados de maneira adequada e da inexistência de capacitação do pessoal envolvido. As organizações possuem problemas por uma falta de gestão de processos, que é característica típica da imaturidade em relação ao nível de seus processos internos e que são bem conhecidos pela indústria. No entanto, a implementação de sistemas de gestão eficientes e corretos com normas, como, por exemplo, da ISO 9000, ISO/TS 16.949, podem diminuir significativamente alguns destes tipos de problemas. Zabeu et al. (2006) ainda destaca:

- Cronogramas não observados e, consequentemente, prazos de projetos em níveis de 50% de não atendimento.
- 25% dos projetos de *software* falham ou são abandonados pela existência de dificuldades ou erros.
- Módulos de *software* não funcionam corretamente quando interagem entre si.
- Programas que não fazem o que era esperado ou com uma eficácia não adequada ao usuário.
- Programas difíceis de serem utilizados e entregues com defeitos (15% dos defeitos permanecem no produto entregue ao cliente).
- 30% a 44% do tempo são utilizados para retrabalho nas companhias (tempo não produtivo).
- Programas que param de funcionar sem motivo aparente.

1.4 Análise econômica e de requisitos

A análise de requisitos possui alguns pontos importantes envolvendo a área de gerenciamento de projetos, pois tem como responsabilidade relacionar as exigências e necessidades do cliente e propor atividades para atingir seus objetivos por meio de soluções que deverão ser desenvolvidas e entregues em um produto final de *software*. Desta forma, a análise de requisitos abrange o entendimento, ou seja, os estudos das necessidades que o usuário necessita e solicita para que sejam criadas soluções. Esta análise é determinante para o sucesso ou fracasso do projeto.

Basicamente, a análise de requisitos envolve o reconhecimento do problema, sua avaliação, uma abordagem para a solução, sua modelagem, a especificação dos requisitos e a sua validação.

Na análise de requisitos, é importante ter definidos os conceitos de requisitos do projeto, requisitos do produto, requisitos funcionais e os não funcionais.

De acordo com Sommerville (2006), os **requisitos funcionais** "são requisitos diretamente ligados à funcionalidade do *software*, descrevem as funções que o *software* deve executar", como, por exemplo:

- O sistema deverá permitir o cadastro de produtos;
- O sistema deverá ter um carrinho de compras no bloco lateral;
- O sistema deverá permitir o pagamento por meio de cartão de crédito e boleto bancário.

Os **requisitos não funcionais**, ainda de acordo com Sommerville (2006), "expressam condições que o *software* deve atender ou qualidades específicas que o *software* deve ter". Ou seja, os requisitos não funcionais "colocam restrições no sistema". Podemos ainda destacar que os requisitos não funcionais são os requisitos relacionados ao uso da aplicação em termos de desempenho, usabilidade, confiabilidade, segurança, disponibilidade, manutenibilidade e tecnologias envolvidas. Exemplos de requisitos não funcionais:

- No sistema, não poderão ocorrer perdas de informações;
- O sistema deverá comportar com velocidade satisfatória, não superando o tempo de 5 segundos para carregamento de páginas que hajam consultas;
- O sistema deverá ser compatível com os principais navegadores.

A Análise de Requisitos é uma tarefa que envolve, em primeiro lugar, um trabalho de descoberta, refinamento, modelagem e especificação das necessidades e desejos para o *software* a ser desenvolvido. Nesta tarefa, tanto o cliente quanto o desenvolvedor irão desempenhar um papel importante, uma vez que será o cliente que irá efetuar a formulação (concretamente) dos pontos necessários em termos de funcionalidades e desempenho, enquanto que o desenvolvedor atuará como investigador, consultor e será o responsável pela solução destes problemas levantados (MAZZOLA, 2010, p. 63).

De acordo com Pressman (2011, p. 151), os resultados da análise de requisitos na especificação das características de funcionamento de *software* "indicam a interface do *software* com outros elementos do sistema e estabelecem restrições que o *software* deve atender". Eles também permitem que o engenheiro de *software*, analista ou modelador, desenvolva as necessidades mais básicas estabelecidas durante as tarefas que envolvem algumas atividades da engenharia de requisitos, como concepção, levantamento e negociação.

A etapa de análise de requisitos é caracterizada principalmente através da realização de um conjunto de tarefas, que serão discutidas nas seções seguintes.

1.4.1 Análise do problema

A tarefa de análise do problema envolve o estudo dos documentos obtidos e levantados na especificação do sistema a ser desenvolvido, bem como a análise do plano do *software*, com o intuito de compreender as questões acerca do *software* no sistema e checar o escopo que está sendo utilizado para a definição das estimativas do projeto, tais como custo, prazos e recursos diversos. É a primeira tarefa. O gerente de projetos pode ter uma atuação na coordenação dos contatos e fazer a ligação necessária entre o analista e a equipe por parte do cliente. Esta comunicação é fundamental para o projeto e para a análise inicial. Desta forma, o analista tem como principal objetivo neste momento avaliar os problemas das eventuais detecções, sob o ponto de vista do cliente (MAZZOLA, 2010, p. 63).

1.4.2 Avaliação e síntese

A segunda tarefa trata da análise dos fluxos de informação, bem como a elaboração das funções de tratamento e aspectos relacionados ao comportamento do *software*. Também envolve as questões relacionadas à interface com o sistema e as especificações de restrições do projeto (MAZZOLA, 2010, p. 63).

Desta forma, o analista poderá iniciar, após finalizada a análise, a síntese das soluções para os problema. O analista deverá considerar, nesta síntese, as estimativas e as restrições do projeto. Este é um processo que se mantém até que analista e cliente estejam em sintonia sobre a adequação das especificações (MAZZOLA, 2010, p. 63).

1.4.3 Modelagem

Com o objetivo de obter um melhor entendimento dos fluxos de informação e de controle e dos aspectos funcionais e de comportamento, o analista poderá, a partir da tarefa de avaliação e síntese, estabelecer um modelo do sistema (MAZZOLA, 2010, p. 64).

Para reforçar o conhecimento sobre a viabilidade do *software* a ser desenvolvido, pode ser importante a criação de um protótipo de *software* como alternativa à análise de requisitos.

Ao criar o modelo de análise, há algumas importantes regras práticas a serem seguidas, conforme sugere Pressman (2011, p. 152):

- O modelo deve enfocar as necessidades visíveis no domínio do problema ou do negócio. O nível de abstração deve ser relativamente elevado. "Não se perca nos detalhes" que tentam explicar como o sistema irá funcionar.
- Cada elemento do modelo de requisitos deve contribuir para o entendimento geral dos requisitos de software e fornecer uma visão do domínio de informação, função e comportamento do sistema.
- Postergue considerações de infraestrutura e outros modelos não funcionais até a fase de projeto. Ou seja, talvez seja preciso um banco de dados, porém as classes necessárias para sua implementação, as funções necessárias para acessar o banco de dados e o comportamento que será apresentado à medida que ele for usado devem ser considerados apenas depois da análise do domínio do problema ter sido completada.
- Minimize o acoplamento do sistema. É importante representar os relacionamentos entre as classes e funções. Entretanto, se o nível de "interconexão" for extremamente alto, deve-se esforçar para reduzi-lo.
- Certifique-se de que o modelo de requisitos agrega valor a todos os interessados. Cada participante tem um uso próprio para o modelo. Por exemplo, os interessados no negócio devem usar o modelo para validar os requisitos; os projetistas devem usar o modelo como base para o projeto; o pessoal da Garantia da Qualidade (QA) deve usar o modelo para ajudar no planejamento de testes de aceitação.
- Mantenha o modelo o mais simples possível. Não crie diagramas adicionais quando não acrescentam nenhuma informação nova. Não utilize formas de notação complexas, quando uma lista simples já bastaria.

1.4.4 Especificação dos requisitos e revisão

A etapa de análise de requisitos se encerra com a criação de um documento de Especificação de Requisitos de *software*. Este documento tem como objetivo registrar os resultados das tarefas cumpridas. Adicionalmente pode ser desenvolvido um Manual Preliminar do Usuário, que permite que o cliente consiga revisar os requisitos ainda em estágio inicial ao processo de desenvolvimento de *software*, diminuindo as possíveis frustrações quanto a uma má definição com relação aos aspectos do *software* (MAZZOLA, 2010, p. 64).

1.5 Especificação do *Software*

A especificação do *software* tem como objetivo definir quais são as funções requeridas pelos sistemas, além das restrições sobre a operação e o desenvolvimento do mesmo. A esta atividade dá-se o nome de **engenharia de requisitos**. Considera-se a engenharia de requisitos como uma etapa bastante importante do processo de *software*. Erros nesse estágio tornam-se problemas em seguida, no projeto e na implementação do sistema (SOMMERVILLE, 2006, p.46).
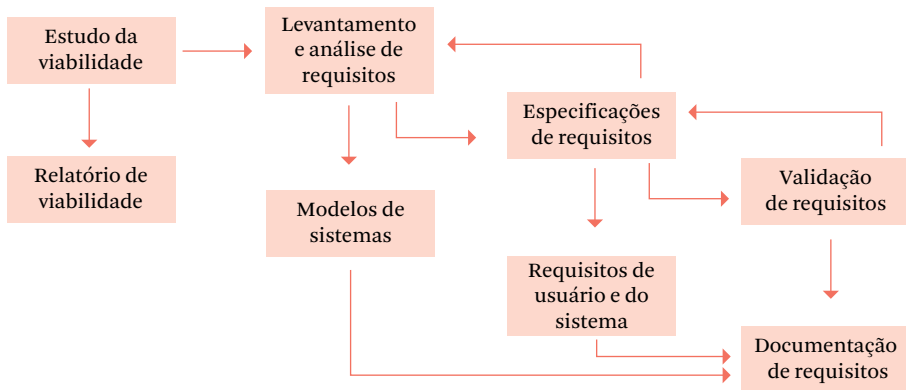
Figura 1.1 – O processo de engenharia de requisitos. Fonte: SOMMERVILLE (2006, p.46) (Adaptado pelo autor).

A figura 1.1 acima mostra o processo de engenharia de requisitos, em que há a produção de uma série de documentação de requisitos, gerando a especificação para o sistema.

Geralmente, na documentação dos requisitos, estes são descritos com diferentes níveis de detalhes, sendo que para os usuários finais e clientes os requisitos são considerados de alto nível, representando a ideia e a forma mental que estes o descrevem, enquanto que para os desenvolvedores os requisitos possuem maior detalhamento e especificações técnicas referentes ao sistema.

Segundo Sommerville (2006, p.47), são quatro as principais fases no processo de engenharia de requisitos, como pode ser visto a seguir:

1. ESTUDO DE VIABILIDADE	É feita uma estimativa para verificar se as necessidades dos usuários que foram identificadas podem ser satisfeitas com a utilização das atuais tecnologias de *software* e *hardware*. O estudo decidirá se o sistema proposto será viável, do ponto de vista comercial, e se poderá ser desenvolvido considerando certas restrições orçamentárias existentes. Um estudo de viabilidade deve ser relativamente barato e rápido. O resultado deve informar se a decisão é a de prosseguir com uma análise mais detalhada.
2. LEVANTAMENTO E ANÁLISE DE REQUISITOS	Este é o processo de obter os requisitos do sistema pela observação de sistemas existentes, pela conversa com usuários e compradores em potencial, pela análise de tarefas e assim por diante. Ele pode envolver o desenvolvimento de um ou mais diferentes modelos e protótipos de sistema. Isso ajuda o analista a compreender o sistema a ser especificado.
3. ESPECIFICAÇÃO DE REQUISITOS	É a atividade de traduzir as informações coletadas durante a atividade de análise em um documento que defina um conjunto de requisitos. Dois tipos de requisitos podem ser incluídos nesse documento. Os requisitos dos usuários são declarações abstratas dos requisitos de sistema para o cliente e os usuários finais do sistema: os requisitos do sistema são uma descrição mais detalhada da funcionalidade a ser fornecida.
4. VALIDAÇÃO DE REQUISITOS	Essa atividade verifica os requisitos quanto à sua pertinência, consistência e integralidade. Durante esse processo, inevitavelmente são descobertos erros na documentação de requisitos. Os requisitos devem então ser modificados, a fim de corrigir esses problemas.

Não há uma sequência rígida para as atividades no processo de requisitos. Há simultaneidade, pois a análise de requisitos continua durante a definição e a especificação. Além disso, novos requisitos podem surgir no processo (SOMMERVILLE, 2006, p.47).

A especificação, segundo Medeiros (2014), deve ser clara e demonstrar as necessidades do cliente, podendo ser um documento escrito, um protótipo,

modelos, gráficos ou cenários. Há uma flexibilidade nesta escolha de como documentar a especificação. A ideia é atender à necessidade do projeto, que quando grande demanda uma documentação escrita e gráfica; e quando menor pede, muitas vezes, apenas cenários de casos de usos.

A especificação de requisitos de *software* é uma descrição mais detalhada de todos os aspectos do *software* a ser construído, elaborada antes do início do projeto. Medeiros (2014) propõe, a seguir, um modelo para essa especificação, dividido em tópicos:

Sumário
Histórico de Revisão
- 1. Introdução
  - 1.1. Propósito
  - 1.2. Convenções do documento
  - 1.3. Público-alvo e sugestão de leitura
  - 1.4. Escopo do projeto
  - 1.5. Referências
- 2. Descrição geral
  - 2.1. Perspectiva do produto
  - 2.2. Características do produto
  - 2.3. Classes de usuários e características
  - 2.4. Ambiente operacional
  - 2.5. Restrições de projeto e implementação
  - 2.6. Documentação para usuários
  - 2.7. Hipóteses e dependências
- 3. Características do sistema
  - 3.1. Características do sistema 1
  - 3.2. Características do sistema 2
  - 3.3. Características do sistema n
- 4. Requisitos de interfaces externas
  - 4.1. Interfaces do usuário
  - 4.2. Interfaces de *hardware*
  - 4.3. Interfaces de *software*
  - 4.4. Interfaces de comunicação
- 5. Outros requisitos não funcionais
  - 5.1. Necessidades de desempenho
  - 5.2. Necessidades de proteção
  - 5.3. Necessidades de segurança
  - 5.4. Atributos de qualidade de *software*
- 6. Outros requisitos

Apêndice A: Glosário
Apêndice B: Modelos de análise
Apêndice C: Lista de problemas

Modelo de um documento de especificação. MEDEIROS, 2014.

A avaliação dos artefatos produzidos na engenharia de requisitos é o processo da validação. Neste processo é avaliada a qualidade dos artefatos produzidos, verificando a especificação. Assim, verifica-se a clareza dos requisitos declarados, destacando possíveis inconsistências e erros. Avalia-se também se os artefatos estão dentro de um padrão definido para o processo e para o projeto (MEDEIROS, 2014).

A principal forma de validação ocorre por meio da revisão técnica. Desenvolvedores, clientes e usuários examinam a especificação em busca de equívocos, destacando a necessidade de maiores esclarecimentos ou informações (MEDEIROS, 2014).

A Gestão de Requisitos é realizada por uma equipe, que tem como objetivo identificar, controlar e acompanhar as necessidades e as mudanças a qualquer momento no projeto(MEDEIROS, 2014).

1.6 Desenho ou arquitetura do sistema de *software*

A **arquitetura de *software*** refere-se à "organização geral do *software* e aos modos pelos quais disponibiliza integridade conceitual para um sistema".

> Em sua forma mais simples, arquitetura é a estrutura ou a organização de componentes de programa (módulos), a maneira pela qual esses componentes interagem e a estrutura de dados usada pelos componentes. Em um sentido mais amplo, entretanto, os componentes podem ser generalizados para representar os principais elementos de um sistema e suas interações (PRESSMAN, 2011, p.213).

Um objetivo do projeto de *software* é desenvolver um quadro da arquitetura de um sistema. Um conjunto de padrões de arquitetura e reuso de *software* para problemas semelhantes (PRESSMAN, 2011, p.213).

Shaw e Garlan [Sha95a] detalham as propriedades que devem ser especificadas como parte de um projeto de arquitetura (PRESSMAN, 2011, p.213):

Propriedades estruturais. Esse aspecto do projeto da representação da arquitetura define os componentes de um sistema (por exemplo, módulos, objetos, filtros) e a maneira pela qual os componentes são empacotados e interagem entre si. Por exemplo, objetos são empacotados para encapsularem dados e o processamento que manipula esses dados e interagem por meio da chamada dos métodos (PRESSMAN, 2011, p.213).

Propriedades não funcionais. A descrição do projeto de arquitetura deve descrever a maneira pela qual o projeto da arquitetura atinge os requisitos de desempenho, capacidade, confiabilidade, segurança, adaptabilidade e outras características do sistema, que não representam as funcionalidades diretamente acessadas pelos usuários (PRESSMAN, 2011, p. 213).

Famílias de sistemas. O projeto de arquitetura deve tirar proveito de padrões reusáveis comumente encontrados no projeto de famílias de sistemas similares. Em essência, o projeto deve ter a habilidade de reutilizar os componentes que fazem parte da arquitetura (PRESSMAN, 2011, p. 213).

Os principais aspectos do funcionamento de um *software* embasam o conceito de arquitetura de *software*, sendo eles a estrutura hierárquica de seus componentes e as estruturas de dados. Para Mazzola (2010, p. 88), "a arquitetura de *software* resulta do desenvolvimento de atividades de particionamento de um problema, encaminhadas desde a etapa de Análise de Requisitos". Inicialmente, há a definição das estruturas de dados e dos componentes de *software*; e a solução de parte ou de todo o problema ocorre ao longo do projeto, por meio da definição de um ou mais elementos de *software* (MAZZOLA, 2010, p. 88).

Não há técnica de projeto que crie uma solução única. Há diversas soluções para um mesmo conjunto de requisitos de *software*. O desafio está, justamente, em escolher qual é a melhor alternativa para a solução do problema (MAZZOLA, 2010, p. 88).

No processo de análise de requisitos de *software* há a modelagem, que possibilita um melhor entendimento das questões de arquitetura e comportamento do problema a ser resolvido com o auxílio do *software*.

Um modelo realizado durante a etapa de Análise de Requisitos deve concentrar-se na representação do que o *software* deve realizar e não em como ele o realiza.

O modelo deve enfocar aquilo que o *software* deve realizar, e é comum e esperado que os modelos sejam expressos graficamente, trazendo descrições complementares em texto natural ou especializada (MAZZOLA, 2010, p. 66).

A obtenção de um modelo dos requisitos do *software* é útil aos agentes envolvidos no desenvolvimento do *software* (MAZZOLA, 2010, p. 66):

- ao **analista**, para uma melhor compreensão da informação, das funções e do comportamento do sistema, o que pode tornar a tarefa de análise mais sistemática;
- ao **pessoal técnico** como um todo, uma vez que ele pode ser uma referência de revisão, permitindo a verificação de algumas propriedades, como a completude, a coerência e a precisão da especificação;
- ao **projetista**, servindo de base para o projeto por meio da representação dos aspectos essenciais do software.

1.7 Codificação (Implementação)

O nome dado ao conjunto de atividades relacionadas à formalização é Codificação. Nesta etapa é utilizada a linguagem de programação, que traz maior proximidade com a linguagem processada pela máquina (MAZZOLA, 2010, p. 103).

> O estágio de implementação do desenvolvimento de *software* é o processo de conversão de uma especificação de sistema em um sistema executável. Esse estágio sempre envolve processos de projeto e programação de *software*, mas, se uma abordagem evolucionária de desenvolvimento for utilizada, ele poderá também envolver o aperfeiçoamento da especificacão de *software* (SOMMERVILLE, 2006, p. 47).

O código-fonte é um elemento essencial para as atividades de validacão do *software* e para as tarefas de manutencão, sendo considerado, assim, bastante relevante no processo de desenvolvimento de *software*. "O aspecto de documentação é um aspecto que deve ser bastante considerado na etapa de codificação" (MAZZOLA, 2010, p. 109).

1.8 Testes do produto

Apesar de melhorar significativamente o produto, não há garantia de qualidade pelo uso das metologias, técnicas e ferramentas da Engenharia de *Software* no processo de desenvolvimento

Assim, para que haja maior qualidade, é fundamental que a etapa de procedimentos de teste seja realizada, sendo esta a última etapa de revisão da especificação do projeto e da codificação (MAZZOLA, 2010, p. 113).

"A realização, de forma cuidadosa e criteriosa, dos procedimentos associados ao teste de um *software* assume uma importância cada vez maior, dado o impacto sobre o funcionamento (e o custo) que este componente tem assumido nos últimos anos" (MAZZOLA, 2010, p. 113). Assim, a etapa de teste pode chegar a 40% do esforço total empreendido no desenvolvimento do *software*.

Para estas atividades de teste há uma série de regras e objetivos trazidos por Glen Myers (PRESSMAN, 2011, p. 121):

• Teste consiste em um processo de executar um programa com o intuito de encontrar um erro.

• Um bom pacote de testes é aquele em que há uma alta probabilidade de encontrar um erro ainda não descoberto.

• Um teste bem-sucedido é aquele que revela um novo erro.

PRESSMAN, 2011, p. 121.

Encontrar erros é o maior objetivo do teste, sendo que um teste adequadamente bem feito traz alta probabilidade de encontrar um erro. Um engenheiro de *software* deve projetar e implementar um sistema tendo em mente a capacidade deste ser testado. Os testes devem ter uma série de características que permitam atingir o objetivo de encontrar o maior número de erros, como pode ser visto abaixo (PRESSMAN, 2011, p. 429):

Testabilidade – James Bach dá a seguinte definição para testabilidade: "Testabilidade de *software* é simplesmente a facilidade com que um programa de computador pode ser testado". As seguintes características levam a um *software* testável.

Operabilidade – Quanto melhor funcionar, mais eficientemente pode ser testado. Se um sistema for projetado e implementado tendo em mente a qualidade, haverá poucos defeitos bloqueando a execução dos testes, permitindo que o teste ocorra sem sobressaltos.

Observabilidade – O que você vê é o que você testa. Entradas fornecidas como parte do teste produzem saídas distintas. Estados e variáveis do sistema são visíveis ou podem ser consultados durante a execução. Saída incorreta é facilmente identificada. Erros internos são automaticamente detectados e relatados. O código-fonte é acessível.

Controlabilidade – Quanto melhor pudermos controlar o *software*, mais o teste pode ser automatizado e otimizado. Todas as possíveis saídas podem ser geradas por meio de alguma combinação de entrada, e os formatos de entrada e saída são consistentes e estruturados. Todo o código é executável através de alguma combinação de entrada. Estados e variáveis de *software* e *hardware* podem ser controlados diretamente pelo engenheiro de teste. Os testes podem ser convenientemente especificados, automatizados e reproduzidos.

Decomponibilidade – Controlando o escopo do teste, podemos isolar problemas mais rapidamente e executar um releste mais racionalmente. O sistema de *software* é construído a partir de módulos independentes que podem ser testados de forma independente.

Simplicidade – "Quanto menos tiver que testar, mais rapidamente podemos testá-lo." o programa deverá ter simplicidade funcional (por exemplo, o conjunto de características é o mínimo necessário para satisfazer os requisitos); simplicidade estrutural (por exemplo, a arquitetura é modularizada para limitar a propagação de falhas), e simplicidade de código (por exemplo, é adotado um padrão de codificação para facilitar a inspeção e a manutenção).

Estabilidade – "Quanto menos alterações, menos interrupções no teste." As alterações no *software* são pouco frequentes, controladas quando elas ocorrem e não invalidam os testes existentes. O *software* recupera-se bem das falhas.

Compreensibilidade – "Quanto mais informações tivermos, mais inteligente será o teste." O projeto arquitetura! e as dependências entre componentes internos, externos e compartilhados são bem compreendidas. A documentação técnica é instantaneamente acessível, bem organizada, específica, detalhada e precisa. Alterações no projeto são comunicadas aos testadores.

PRESSMAN, 2011, p. 429.

Para o processo de teste ser eficaz é preciso arquitetar casos de teste para avaliar o *software* a ser testado. Esta é a uma atividade complexa e muitas vezes pode até se comparar ao esforço dado para projetar o próprio *software* (MAZZOLA, 2010, p. 114).

Existem basicamente dois princípios básicos utilizados em testes de qualquer produto, oriundos da engenharia, sendo um conhecido como teste de caixa preta (*black box*) e outro chamado de teste de caixa branca (*white box*). No primeiro (*black box*), os testes são realizados a fim de demonstrar que suas

funções são completamente operacionais, já que as funções a serem desempenhadas pelo produto são conhecidas. Já nos testes de caixa branca (*white box*), efetua-se os testes para averiguar se todos os componentes do produto estão completamente ajustados e se estão realizando de maneira satisfatória a sua função, focando no desempenho interno do sistema ou do produto. Vamos ver um pouco mais sobre estes testes a seguir:

1.8.1 O *software* e o teste de caixa preta

No teste de caixa preta (*black box*), o foco está no funcionamento do *software*, ignorando-se a partes internas e sua construção e baseando-se nos requisitos funcionais, de acordo com a ótica do usuário. Os testes são concentrados nas funções que o *software* deve desempenhar, de acordo com o seu propósito.

Neste tipo de teste, alguns problemas ou erros podem ser revelados, embora sejam insuficientes para detectar determinados riscos em um projeto de *software*, já que o testador não avalia o processamento interno das entradas no sistema, verificando apenas se estas entradas estão de acordo com as saídas.

As principais técnicas deste tipo de teste são:

- Teste de caso de uso
- Particionamento de equivalência
- Testes baseados em Grafo
- Teste de todos os pares
- Teste de tabela de decisão
- Tabelas de estado de transição
- Análise de valor limite

1.8.2 O *software* e o teste de caixa branca

No teste de caixa branca *(white box)*, o produto de *software* tem suas partes internas, estrutura, peças e componentes examinadas detalhadamente. Por esta análise interna, ele também pode ser conhecido como teste estrutural e o analista pode ainda testar determinadas partes específicas de um componente. Vale ressaltar que, em *software*, a parte interna significa basicamente o código fonte do *software*.

Os testes de caixa branca possuem um enfoque na implementação do *software*, ao contrário dos testes de caixa preta que voltam-se à interface do mesmo.

Como os testes de caixa branca trabalham exaustivamente com os detalhes internos do *software*, diretamente em seu código fonte, os caminhos lógicos definidos por este também são muito testados para avaliar o seu funcionamento e os resultados esperados.

Mazzola (2010, p. 114) aponta que apesar da importância do teste de caixa branca, este não garante totalmente a correção após a realização dos testes desta natureza, devido à diversidade de caminhos lógicos que podem ser utilizados para se chegar à uma solução, representando um grande obstáculo para que se tenha êxito total.

Pode-se fazer testes utilizando estas duas abordagens, com o teste de caixa preta e o teste de caixa branca, onde os enfoques se complementam, testando aspectos relacionados às funcionalidades, à interface e às partes internas, estruturais do sistema ou *software*. Estes testes são conhecidos como testes de caixa cinza (*gray box*) e há diferentes opiniões sobre este tipo de teste, que não e totalmente aceito por todos e também é chamando por alguns autores de teste de integração.

ATIVIDADES

01. Na sua opinião, é realmente necessário usar processos de desenvolvimento de *softwares* bem estruturados e de qualidade ou daria para desenvolver *softwares* sem passar por estas etapas? Por quê?

02. Quais são os problemas mais comuns encontrados nas atividades de processo de desenvolvimento de *software*?

03. Faça uma pesquisa e relacione os principais itens que deveriam constar em um documento de especificação de *software*.

04. Podemos dizer que desenvolvimento de *software* e codificação (implementação) são a mesma coisa? Por quê?

05. Qual a importância dos requisitos para o desenvolvimento de um *software*? Quais são os principais tipos de requisitos?

06. Assinale a alternativa que **não** corresponde à uma das etapas do processo de desenvolvimento de *software*.

a) Concepção
b) Desenvolvimento
c) Análise
d) Padronização
e) Teste

REFLEXÃO

Neste capítulo, vimos que para desenvolver um *Software* são necessárias muitas etapas, que devem ser realizadas por meio de métodos e processos bem estruturados.

Assim como é utilizado para a construção civil ou em projetos de produtos manufaturados, o *Software* também passa por processos e análise que vão desde a sua concepção até a fases de manutenção, passando basicamente também por planejamento, modelagem, construção (codificação), testes e entrega final.

Por ser uma atividade complexa, é natural que aconteçam alguns problemas, e de fato existem aqueles que são comuns, principalmente causados por falhas ou negligências em alguma das etapas do processo de desenvolvimento do *Software*. Você consegue perceber a quantidade de áreas do conhecimento que são envolvidas em uma atividade de desenvolvimento? Note que usamos áreas da engenharia e seus processos, gerenciamento de projetos, gestão de pessoas, métricas e estimativas de custo, recursos e tempo, além do próprio conhecimento técnico aplicado com linguagens de programação e pilhas de tecnologias diversas, bem como abordagens de treinamento, capacitação, estratégias de suporte técnico, documentação e um conjunto de alinhamento do *Software* com a regra de negócio do qual ele (*Software*) se propõe a auxiliar ou a atuar.

Embora todos utilizemos vários *Softwares*, sistemas e aplicações há um tempo, no computador, que tal a partir de agora você começar a refletir ou imaginar a complexidade que eventualmente tenha sido para conceber o produto do qual você utiliza atualmente? Como, por exemplo, os seus *Softwares* editores de textos, editores de planilhas eletrônicas, seu navegador de Internet, seus *sites*, portais ou redes sociais de preferência … e assim por diante. Imagine como foi todo o processo de desenvolvimento destes *softwares*, e se for necessário, dê uma pesquisada também. Será bem interessante!

LEITURA

O livro de Ian Sommerville, 9.ed., 2011, faz uma excelente abordagem sobre processos de *software* e requisitos de *software* e sua consulta vale muito a pena.

É recomendável a leitura do artigo "Melhoria de Processos de Desenvolvimento de *software* Aplicando Process Patterns", de autoria de Paulo Augusto O. Tamaki e Kechi Hirama, onde eles propõem um método para melhoria de processos aplicando process patterns, com o propósito de preenhcer uma lacuna entre a estrutura metodológica do projeto de *software* e o processo de melhorias.

A leitura de capítulos relacionados aos assuntos abordados da parte I do livro de Pressman, 2011, "Engenharia de *Software*, Uma abordagem profissional", também é outra referência que não podemos deixar de fora e que muito contribui para o aprofundamento dos assuntos colocados ao longo do capítulo.

REFERÊNCIAS BIBLIOGRÁFICAS

COSTA, T. M., **Melhoria Contínua de Processo de Software utilizando a teoria das restrições**. Dissertação de Mestrado UFRJ. Rio de Janeiro, 2012. p.233.

FUGGETTA, A., 2000, "**Software process: a roadmap**", pp. 25-34, Limerick, Irlanda.

LEITE, J. C.; **O Processo de Desenvolvimento de Software**. 2000. Disponível em: <http://www.dimap.ufrn.br/~jair/ES/c2.html>. Acesso em: 10 dez. 2014.

MARCORATTI. **O processo de Software**, 2014. Disponível em: <http://www.macoratti.net/proc_sw1.htm>. Acesso em: 28 nov. 2014.

MAZZOLA, V. B. **Engenharia de Software - Conceitos Básicos**, Apostila, 2010. Disponível em: <https://jalvesnicacio.files.wordpress.com/2010/03/engenharia-de-software.pdf>. Acesso em: 17 dez. 2014.

MEDEIROS, H. **As Etapas da Engenharia de Requisitos**, 2014. Disponível em: <http://www.devmedia.com.br/as-etapas-da-engenharia-de-requisitos/30220>. Acesso em: 02 dez. 2014.

PAULK, M., CURTIS, B., CHRISSIS, M., et al., 1993, "**Capability maturity model, version** 1.1", IEEE software, v. 10, n. 4, pp. 18-27.

PRESSMAN, R. S. **Engenharia de Software**, 7.ed. São Paulo: McGraw-Hill, 2011. p. 780.

ROCHA, A. R. C.; MALDONADO, J. C.; WEBER K. C. **Qualidade de Software: Teoria e Prática**. 1. ed. Editora Prentice Hall. São Paulo. 2001.

SILVA, B. C. C.; **Processos e Ferramentas para o Desenvolvimento de Software Livre: Um estudo de caso**. Mestrado; UFES; 2006; p. 93.

SOMMERVILLE, I. **Engenharia de Software**, 6. ed. Editora Pearson do Brasil, 2003. p. 292.

SOMMERVILLE, I. **Engenharia de Software**, 6. ed. Editora Pearson do Brasil, 2003. p. 292.

WAZLAWICK, R. S. **Engenharia de Software**: Conceitos e Práticas, 1ª Edição, Elsevier, 2013. p. 506.

ZABEU, C. P., JOMORI, S. M., VOLPE, R. L. D. **A importância da qualidade no desenvolvimento de software**. Banas qualidade. 2006.

2

Suporte e Manutenção do *Software*

Este capítulo aborda os assuntos envolvidos na tríade Suporte-Manutenção -Evolução do *software*. A documentação de *software* está presente em praticamente todas as fases das atividades dos processos de desenvolvimento de *software* e é essencial para prover manutenção com mais eficiência e condições para um suporte técnico de qualidade.

Manter um *software* atualizado com seus requisitos implantados alinhados com o negócio ao qual se propõe é um grande desafio. A manutenção de *software*, além de reparar, ajustar, também garante melhorias constantes com o objetivo de prevenir falhas e buscar um estado de perfeição, quando possível. Estas ações deixam o *software* sempre útil às necessidades do negócio, estendendo o seu uso e aumentando o seu ciclo de vida.

Com a documentação atualizada, presente e completa, as atividades de manutenção e suporte técnico são viabilizadas com maiores chances de sucesso, auxiliando, ainda, na melhora contínua do *software,* o que garantirá maior qualidade do *software*.

Vamos ver e estudar como estes assuntos estão relacionados e compreender que todas as atividades do processo de desenvolvimento de *software* são importantes para a documentação, trazendo maior eficiência no suporte, manutenção e na melhora contínua.

OBJETIVOS

- Aprender e conhecer sobre a importância da documentação de *software*;
- Aprender sobre manutenção de *software* e seus tipos;
- Ter uma visão geral a respeito do suporte e treinamento na área de *softwares*;
- Conhecer sobre os processos de melhoria contínua.

2.1 Introdução

Em 1965, Mark Halpern introduziu o conceito de evolução do *software* para descrever as características de crescimento do mesmo. Mais tarde, o termo "evolução", no contexto de *software* de aplicação, foi amplamente utilizado. O conceito atraiu ainda mais as atenções dos investigadores após Belady e Lehman publicarem um conjunto de princípios que determinam a evolução dos sistemas de *software*. Os princípios eram de natureza muito geral. Em seu artigo intitulado "'*The Maintenanc*e 'Iceberg'," RG Canning comparou a manutenção de *software* a um "iceberg" para enfatizar o fato de que os desenvolvedores de *software* e o pessoal de manutenção enfrentam um grande número de problemas. Alguns anos mais tarde, em 1976, Swanson introduziu o termo "manutenção», agrupando as atividades de manutenção em três categorias básicas: corretiva, adaptativa e perfectiva (TRIPATHY e NAIK, 2015, p. 1).

Manutenção é o termo para qualquer esforço que é colocado em uma parte do *software* após ele ter sido desenvolvido e colocado em operação. Ou, ainda, podemos definir como sendo o processo de melhoria e otimização de um *software* já desenvolvido, visando a correção de defeitos que este venha a ter eventualmente.

A manutenção de *software* é uma atividade importante no ciclo de vida do *software* e possui certas complexidades, além de ter um impacto significativo no custo, de maneira elevada. Como a manutenção trata de imprevisibilidades, há muitas atividades não estruturadas ou não esperadas, mas é de fundamental importância que também haja uma série de atividades estruturadas a fim de garantir a qualidade do *software*, por meio das ações de manutenção preventiva. Atividades corretivas envolvem ações nas falhas que são descobertas, na correção destas falhas ou erros, na acomodação de novos requisitos que surgem ao longo do tempo, mesmo após o *software* estar sendo utilizado em produção, devendo ainda buscar simplicidade, eficiência e um conjunto de atualizações tecnológicas que envolvem este produto de *software*.

Há uma quantidade razoável de documentação em *softwares* desenvolvidos, envolvendo documentos de trabalhos, manuais de usuários e outros artefatos que são gerados, em sua maioria, por engenheiros de *software*.

A documentação do *software* é também de extrema importância para garantir a qualidade do produto desenvolvido, em toda a sua extensão do ciclo de vida, promovendo uma manutenção eficiente. Quanto mais completa a

documentação e com artefatos diversos gerados em todas as fases do desenvolvimento do *software*, maior será a chance de garantir também um suporte técnico eficaz e de qualidade. Como o suporte técnico lida com uma diversidade de chamados e situações específicas no dia a dia, ter uma documentação disponível para prestar o serviço de forma segura e pontual é essencial para o sucesso nas resoluções dos chamados abertos. Além do uso da documentação em suporte técnico, esta também contribui para os treinamentos e capacitações que são realizados, podendo usar artefatos como, por exemplo, manuais de uso e documentos voltados para o usuário.

Assim, podemos notar que a manutenção de *software*, aliada a uma boa documentação e a um suporte técnico de qualidade e eficiente, irão contribuir diretamente na melhoria contínua do *software*, melhorando cada vez mais os processos envolvidos e mitigando problemas e erros que foram descobertos e resolvidos ao longo da manutenção.

Vamos estudar um pouco mais sobre estes assuntos a seguir.

2.2 Documentação

A documentação de *software* está diretamente relacionada à manutenção, suporte técnico, treinamento e melhora contínua de um sistema. É fundamental que haja uma documentação completa e eficiente para que haja maior segurança e conforto nas práticas de manutenção, já que ela se estende por todas as fases do processo de desenvolvimento do *software*. Independentemente da etapa do ciclo de vida de um sistema, a documentação pode estar presente por meio de seus diferentes artefatos, como notações, documentos, modelagens, *workflows* e qualquer outro meio que tenha a função de documentar aquela etapa, um processo, uma atividade, comportamento ou especificações técnicas.

Imagine um sistema desenvolvido sem qualquer tipo de documentação. Neste momento, vamos tentar excluir o código-fonte desta situação, onde muitos desenvolvedores fazem comentários em meio ao código e acreditam ser isto a documentação do *software*. E então, quais seriam as consequências de um sistema desenvolvido sem nenhum artefato gerado com o propósito de documentação? Para tentar expandir ainda mais a imaginação, vamos fazer um paralelo com a engenharia civil, em um cenário onde precisássemos construir uma grande obra física, como uma casa, um prédio ou até mesmo construções

mais elaboradas como um robusto *shopping center*. Será que daria certo a construção sem nenhuma planta (elaborada por arquitetos), sem nenhuma especificação ou sem nenhum documento que pudesse ser seguido? Ou, ainda, por mais que contássemos com a grande experiência de alguns profissionais, que supostamente fariam este trabalho sem nenhum documento ou planta para se apoiarem, como seria a manutenção anos depois nesta obra, realizada por outros profissionais ou até mesmo por outra empresa? Em uma eventual reforma, como os profissionais saberiam ao certo e com precisão os locais exatos em que os canos, tubulações e fiação estão sendo passados nas paredes desta construção? Ou, ainda, os materiais que foram utilizados e seus fornecedores, modelos e especificações? Veja que, por meio desta analogia, é possível imaginar como também poderia acontecer com a engenharia de *software*, no desenvolvimento de sistemas complexos. A documentação de *software* é de extrema importância e necessária para que todas as especificações sejam desenvolvidas do modo como é esperado e para que todas as ações e detalhes realizados ao longo do processo de desenvolvimento sejam devidamente anotados e estruturados em documentos e/ou artefatos diversos.

Os requisitos são os principais elementos que norteiam o desenvolvimento de sistemas e sua especificação clara e objetiva é fundamental para definição do *software*. Desta forma, a documentação surge desde esta especificação dos requisitos, documentos de negociações com o cliente, modelagens, anotações, manuais, roteiros, guias e tudo o que for gerado durante todo este grande processo de desenvolvimento. O detalhamento dos requisitos e sua clareza irão impactar diretamente na qualidade do produto final.

O propósito de um projeto é outro fator determinante para avaliarmos a obrigatoriedade dos artefatos que deverão ser gerados e os seus níveis de detalhamentos. Assim, vale ressaltar que, como os requisitos sofrem constantes alterações sob influência dos próprios clientes, esta documentação de requisitos não pode ser considerada algo como o contrato com o cliente, pois muitas vezes estes clientes também a ignoram e impõe suas próprias vontades, fazendo as equipes cederem. O contrato é um instrumento legal que acorda entre as partes vários pontos referentes aos serviços e produto que será desenvolvido, enquanto que uma documentação busca formalizar definições, decisões referentes às especificações dos requisitos e outros detalhes referentes ao desenvolvimento do *software*.

Fagundes (2011, p. 24) faz uma análise com os números extraídos das estimativas de esforço aplicado em manutenção e traz uma abordagem interessante. Veja no quadro a seguir.

"Segundo as estimativas (confiáveis ou não) mais disseminadas no mercado, cerca de 15 a 20% do esforço de um projeto são consumidos com a atividade de requisitos. Isso já é um alerta, pois acredito que é tempo demais e que, provavelmente, há muitos fatores geradores de baixa produtividade: falta de treinamento, de participação, de compreensão, de fontes de informação etc. No restante do esforço gasto, 80%, numa análise pessimista, a documentação está disponível (e provavelmente em uso) para a equipe de desenvolvimento consultar .

Imaginemos que desses 20% em requisitos, 80% sejam gastos efetivamente com reuniões, leituras, contatos por telefone ou e-mail, elaboração de documentação e protótipo e revisão. A validação com o cliente ocuparia 20% do tempo total em requisitos. Aqui cabe tecer alguns comentários:

- A proporção 80-20 é irreal, pois proporcionalmente gasta-se muito mais esforço investigando do que validando, mas para efeito da brincadeira numérica, vale pensar num cenário mais pessimista;
- O valor definido é em termos de esforço efetivo – não do tempo que a documentação passa com o cliente sem sequer ser aberta para leitura e;
- Muitas vezes a documentação fica bastante tempo com o cliente e, ao final, é aprovada sem que se faça de fato a validação, sendo a aposição da assinatura um gesto pró-forma diante da proximidade do prazo.

Considerando também que nem todo o esforço consumido pelo restante da equipe é feito consultando a documentação, imaginemos que em 30% do trabalho seja feita alguma consulta ou que o labor seja conduzido pela documentação. É uma proporção razoável, se imaginarmos que uma boa parte da equipe usa de forma intensiva a documentação: arquitetos, gerente de projeto, analista de testes, *designer*, implementador...

Diante dessa brincadeira numérica, descobre-se que o cliente consumiria 4% do esforço acompanhado da documentação, enquanto a equipe de desenvolvimento, excluindo-se o próprio analista de requisitos, 24%, seis vezes mais. Se incluirmos o analista de requisitos, esse percentual subiria para 40%, representando 10 vezes maior uso dentro da equipe de desenvolvimento do que para o cliente. Sem mencionar o problema da percepção divergente, com o cliente ignorando a documentação em momentos que deseja uma redefinição de requisitos" (FAGUNDES, 2011, p. 25).

A documentação descreve várias partes do código-fonte, como, por exemplo, uma função, uma classe ou módulo. Nela consta um conjunto de formas de notações que envolvem manuais gerais e de ordem técnica, organizados textualmente, por meio de comentários, dicionários, diagramas, *workflows,* diagramas, modelagens diversas que podem ser representadas por gráficos e desenhos, entre outros. É ainda necessário efetuar um planejamento, identificando os documentos que serão gerados, definindo de sua organização e relacionamento entre os outros documentos, bem como definindo a linguagem que será utilizada (COELHO, 2009, p. 2).

Uma documentação de *software* deve considerar alguns pontos importantes, como a comunicação entre a estrutura do sistema e o seu comportamento; visualização e controle da aquitetura do sistema; apresentar possibilidades de simplificação e reutilização; e gerenciar adequadamente os riscos.

Coelho (2009, p. 2) menciona os dois tipos básicos de documentação, propostos por Michelazzo (2006), por meio de uma tabela que representa a documentação técnica e a documentação de uso.

DOCUMENTAÇÃO TÉCNICA	DOCUMENTAÇÃO DE USO
▪ Voltada ao desenvolvedor. ▪ Compreende dicionários e modelos de dados, fluxogramas de processos e regras de negócios, dicionários de funções e comentários de códigos.	▪ Voltada para o usuário final e o administrador do sistema ▪ Formada por apostilas ou manuais que apresentam como o manual deve ser usado, o que esperar dele e como receber as informações que se deseja

Fonte: COELHO (2009, p. 3).

Wazlawick (2013, p. 21) apresenta um pequeno modelo de documento para atividades de um processo com as seguintes seções:

a) Cabeçalho:

- Nome do processo.
- Nome da fase (se cabível).
- Nome da atividade.
- Versão do documento.

- Responsável (cargo).
- Participantes (opcional).
- Artefatos de entrada (opcional).
- Artefatos de saída.
- Recursos alocados (opcional).

b) Corpo contendo o detalhamento da atividade.

A documentação de *software* pode unir uma série de documentos e materiais como, por exemplo: Documento de requisitos, Documento descritivo da arquitetura do sistema, e de cada um dos programas, listagem do código-fonte do *software*, documentos de validação e seus relacionamentos, além de guias de manutenção com os problemas já identificados, entre outros. Estes documentos também podem ser conhecidos como artefatos.

2.2.1 Artefatos

Conforme mencionamos acima, em uma documentação de *software* podem existir vários artefatos, que são gerados ao longo do processo. Os artefatos são quaisquer documentos que foram gerados ou produzidos ao longo de todo o processo de desenvolvimento de *software*, como diagramas, programas, documentos de textos, anotações, contratos, especificações de projeto, desenhos, modelos, projetos, planos etc (WAZLAWICK, 2013, p. 21).

Em alguns modelos de processo de desenvolvimento, ou metodologias, como, por exemplo, no Processo Unificado (UP), apenas o autor que criou o artefato pode efetuar alterações em seu respectivo artefato, trazendo maior controle e evitando duplicações de informações. Em outros modelos como o XP, por exemplo, esta prática é inversa, ou seja, os artefatos não possuem um único dono e todos podem contribuir com modificações nos artefatos gerados por outros, desde que se tenha verdadeiros motivos para isto. De qualquer modo, é muito importante que estes artefatos estejam em um ambiente dotado de controle de versão para garantir o controle das eventuais mudanças que podem ocorrer de maneira indevida e reverter à versão correta.

Algumas considerações sobre os artefatos de *software*:

• A quantidade de artefatos pode variar de projeto para projeto, dependendo das complexidades envolvidas.

• Alguns artefatos podem não ser guardados para sempre, devido à maneira de como tenham sido criados, como, por exemplo, o próprio código-fonte, que pode sofrer mudanças.

• Também podem ser chamados de "*Deliverables*", sendo que é o que deve ser produzido, concretamente, pelas atividades do processo.

Sauvé (2001) classifica e relaciona alguns artefatos de *software*, como pode ser visto a seguir:

- **Artefatos da fase de elaboração**
    - Documento de *business case* (investigação preliminar)
    - Documento de orçamento e cronograma
    - Documento de especificação de requisitos
        - Requisitos funcionais (Modelo de *use case*)
        - Requisitos não funcionais
    - Modelo conceitual inicial
    - Glossário
    - Projeto arquitetural

- **Artefatos da fase de construção**
    - Refinamento dos requisitos funcionais (Modelo de *use case*)
    - Refinamento do modelo conceitual (Modelo conceitual)
        - Inclui vários tipos de diagramas UML
    - Refinamento do projeto arquitetural
    - Refinamento do glossário
    - Projeto de baixo nível (Modelo de projeto)
        - Pode incluir vários tipos de diagramas UML
        - Pode incluir uma documentação no estilo Javadoc
        - Esquema lógico de banco de dados
    - Testes de unidade e testes de aceitação
    - Código

- **Artefatos da fase de implantação**
    - Planos de publicação
    - Planos de testes alfa, beta
    - Planos de treinamento

2.2.2 Documentação do código-fonte

A documentação do código-fonte é feita basicamente a partir de comentários direto no código-fonte ou por meio de geração de uma documentação on-line. Os comentários ao longo do código-fonte são um assunto que causa grande polêmica, pois para alguns autores isto não pode ser considerado uma documentação, mas é fato que este mecanismo pode ser muito útil aos desenvolvedores, desde que seja utilizado de maneira eficiente, sendo um grande aliado às tarefas ou atividades de manutenção, onde o desenvolvedor pode encontrar comentários pontuais em meio ao código, facilitando a interpretação do *script* ou até mesmo para encontrar determinados trechos específicos que esteja procurando ou para seguir padrões, recomendações e instruções diversas.

```
/**
 * Registers the text to display in a tool tip.   The text
 * displays when the cursor lingers over the component.
 *
 * @param text  the string to display.  If the text is null,
 *              the tool tip is turned off for this component.
 */
public void setToolTipText(String text) {
```

Figura 2.1 – Exemplos de comentários na linguagem de programação JAVA, delimitados por "/*" e "*/". Disponível em: WIKIPEDIA (http://en.wikipedia.org/wiki/Comment_%28computer_programming%29)

```
#!/usr/bin/env python

# this program prints "Hello World" to the screen and then quits.
print("Hello World!")
```

Figura 2.2 – Exemplos de comentários na linguagem de programação Phyton, delimitados por "#". Disponível em: WIKIPEDIA (http://en.wikipedia.org/wiki/Comment_%28computer_programming%29)

Mazzola (2010, p. 109) atenta para a escolha dos identificadores de variáveis e procedimentos como sendo um dos primeiros passos na documentação do código-fonte já que estes necessitam ter um significado claro e explícito para o contexto da aplicação que está sendo considerada. O autor ainda aponta para um outro aspecto importante, que é a formatação do código-fonte . A indentação do código facilita a interpretação do mesmo por parte do desenvolvedor e permite explicitar melhor a combinação dos blocos básicos de uma determinada linguagem de programação. As ferramentas CASE, por meio de IDEs (Ambiente de Desenvolvimento Integrado) possibilitam estas formatações automaticamente.

Na figura 2.3, podemos notar um exemplo do emprego de indentação na linguagem de programação C.

Código em C **com indentação**:

```
if (unlikely(prev->policy == SCHED_RR))
    if (!prev->counter)
    {
        prev->counter = NICE_TO_TICKS(prev->nice);
        move_last_runqueue(prev);
    }
    switch (prev->state)
    {
    case TASK_INTERRUPTIBLE:
        if (signal_pending(prev))
        {
            prev->state = TASK_RUNNING;
            break;
        }
    default:
        del_from_runqueue(prev);
    }
prev->need_resched = 0;
```

O mesmo código, **sem indentação**:

```
if (unlikely(prev->policy == SCHED_RR))
if (!prev->counter)
{
prev->counter = NICE_TO_TICKS(prev->nice);
move_last_runqueue(prev);
}
switch (prev->state)
{
case TASK_INTERRUPTIBLE:
if (signal_pending(prev))
{
prev->state = TASK_RUNNING;
break;
}
default:
del_from_runqueue(prev);
}
prev->need_resched = 0;
```

Figura 2.3 – Um exemplo do emprego de indentação em C. Disponível em: WIKIPEDIA (http://pt.wikipedia.org/wiki/Indenta%C3%A7%C3%A3o)

A documentação ágil

No livro “Modelagem Ágil – Práticas eficazes para a programação extrema”, Ambler (2004, p. 153) aborda sobre a documentação ágil e traz a pergunta: Quando um documento é ágil? Veja abaixo:

“Independentemente do que algumas pessoas lhe dirão, a documentação pode, na verdade, ser muito eficaz. Quando um documento é ágil? Quando satisfaz aos seguintes critérios:

- Documentos ágeis maximizam o retorno dos clientes.
- Documentos ágeis são “magros e econômicos”.
- Documentos ágeis satisfazem a um propósito.
- Documentos ágeis descrevem informações que têm menor probabilidade de mudar.
- Documentos ágeis descrevem “coisas boas de se saber”.
- Documentos ágeis têm um cliente específico e facilitam o trabalho desse cliente.
- Documentos ágeis são suficientemente precisos, consistentes e detalhados.
- Documentos ágeis são suficientemente indexados.”

Em seguida, Ambler (2004, p. 162) lista os pontos importantes relativos à documentação ágil:

- “O ponto fundamental é a comunicação efetiva, não a documentação.
- A documentação deve ser “magra e econômica”.
- Diminua sua carga de trabalho o máximo que puder.
- A documentação deve ser boa apenas o suficiente.
- Os modelos não são necessariamente documentos, e os documentos não são necessariamente modelos.
- A documentação faz parte do sistema tanto quanto o código-fonte.
- O objetivo principal da equipe é desenvolver *software*, o secundário é permitir o próximo trabalho.
- O benefício de ter documentação deve ser maior do que o custo de criá-la e mantê-la.
- Nunca confie na documentação.
- Cada sistema tem suas necessidades de documentação próprias e únicas. Não há uma solução única.
- Pergunte se você PRECISA da documentação e por que acredita que PRECISA dela, não se a quer.
- O investimento na documentação do sistema é uma decisão de negócio, não técnica.
- Crie documentação apenas quando precisar dela – não a crie apenas por criar.
- Atualize a documentação apenas quando necessário.
- O cliente, não o desenvolvedor, determina se a documentação é suficiente.”

AMBLER, 2004, p. 162

2.3 Manutenção de *Software*

Segundo Sommerville (2007), a manutenção é o processo geral de modificação de um sistema depois de ter sido colocado em uso com o objetivo de corrigir defeitos de código, de projeto, de especificação ou acrescentar funcionalidades e quando normalmente não envolve alteração da arquitetura do sistema.

Todo *software* deve ter uma mudança contínua ou se tornará menos útil com a evolução dos sistemas e das organizações. É necessário que o *software* evolua junto, aumentando sua complexidade oferecendo suporte a recursos extras de acordo com as necessidades especificadas pelos usuários e pela evolução das tecnologias.

Podemos dividir a manutenção de *software* em quatro tipos.

CORRETIVA	Realizada para correção de erros não detectados durante o processo de desenvolvimento e testes. Este tipo de manutenção existe porque os testes de *software* dificilmente conseguem detectar todos os erros.
ADAPTATIVA	Realiza alterações que se tornam necessárias por mudanças no ambiente. São necessárias, pois a vida útil dos aplicativos é longa e não acompanha a rápida evolução da computação.
PERFECTIVA OU APERFEIÇOADORA	Visam melhorar o *software* de alguma forma. Geralmente são o resultado de recomendações de novas capacidades bem como modificações de funções existentes solicitadas pelos usuários. Responsável pelo maior esforço gasto com manutenção.
PREVENTIVA	Previne futuras manutenções dos três tipos anteriores. Modificações feitas com o objetivo de melhorar o *software* no que se refere à sua confiabilidade ou manutenibilidade.

A manutenção é a fase mais problemática do ciclo de vida de um *software*. O esforço desprendido nesta etapa pode chegar a 70% de todo ciclo de vida do *software* e os custos podem chegar a 200% do custo de desenvolvimento.

O custo elevado da manutenção é decorrente da dificuldade de entender o que o *software* faz, interpretar as estruturas de dados, as características de interface e limites de desempenho. Além da necessidade de se refazer todas as etapas de desenvolvimento, além de *software* como análise, avaliação, projeto, codificação e teste das manutenções necessárias.

A maioria dos problemas na manutenção de *software* está relacionada à maneira como o *software* foi planejado ou desenvolvido. Quanto maiores forem os problemas e funções deixadas no processo de planejamento e desenvolvimento, maiores serão os problemas encontrados durante a manutenção.

Podemos citar os seguintes problemas apontados por Sommerville (2007), dentre vários existentes:

- dificuldade de acompanhar a evolução do *software* por meio das várias versões. As alterações não são devidamente documentadas;
- dificuldade de acompanhar o processo por meio do qual o *software* foi criado;
- dificuldade de entender programas "de outras pessoas". As "outras pessoas" frequentemente não estão presentes para explicar o que e como foi desenvolvido;
- documentação não existe, é incompreensível ou desatualizada.
- planejamento sem suporte a alterações;

A manutenção de *software* pode ser de dois tipos, estruturada e não estruturada.

2.3.1 Manutenção não estruturada

Neste tipo de manutenção, o único documento disponível sobre o *software* é seu código-fonte. Geralmente, a documentação do código também é falha ou praticamente inexistente. A atividade de manutenção inicia-se com a avaliação do código-fonte do programa em busca do problema ou porção do *software* que sofrerá alteração. Neste caso, o impacto das mudanças é difícil de ser avaliado. E não é possível realizar testes completos que validem todo o sistema com as alterações.

2.3.2 Manutenção estruturada

Neste caso, o *software*, objetivo do trabalho, possui documentação bem estruturada e completa. Assim a atividade inicia-se pela análise da documentação, buscando um entendimento completo do sistema.

Os documentos necessários para uma boa manutenção estruturada são:

- documento sobre os requisitos do sistema;
- documento sobre a arquitetura do sistema;
- especificação e projeto para cada componente do sistema;
- programas-fontes comentados;
- plano de testes.
- guia de manutenção, com os problemas conhecidos.

É possível analisar o impacto das mudanças no sistema antes mesmo da realização da manutenção. Com isso é possível fazer um planejamento adequado, com estimativa de tempo e custos da manutenção.

2.3.3 Manutenibilidade

Manutenibilidade é a facilidade com que um *software* pode ser entendido, corrigido, adaptado, aumentado etc. Em outras palavras, quão fácil é realizar a manutenção deste *software*.

A manutenibilidade é afetada por diversos fatores, entre eles podemos citar:

- negligência nas fases de projeto, codificação e testes;
- instabilidade de pessoal;
- disponibilidade de pessoal qualificado;
- estrutura compreensível do sistema;
- padronização no uso de linguagens de programação e sistemas operacionais;
- padronização das estruturas de documentação;
- disponibilidade de casos de teste;
- disponibilidade de *hardware* adequado;
- qualidade da documentação;

- conhecimento do domínio de aplicação;
- portabilidade.

Não é fácil quantificar a manutenabilidade de um *software*. A principal métrica utilizada para essa medida é o tempo gasto na manutenção, considerando-se nesta medição o tempo de reconhecimento do problema, análise do problema, especificação das mudanças, modificação, testes e o tempo total.

2.3.4 Reengenharia e engenharia reversa

De modo geral, reengenharia é um modo de reusar um *software* e entender os conceitos ocultos do domínio da aplicação. Seu uso facilita isso, pois informações sobre analise e *design* geralmente não estão disponíveis em sistemas legados, que permitam seu reuso.

A reengenharia geralmente é utilizada com um dos seguintes objetivos:

- **melhorar a manutenibilidade**: os esforços com manutenção podem ser minimizados com a reengenharia por produzir módulos menores e mais fáceis de realizar a manutenção. Entretanto, não é possível prever se esse benefício será alcançado;
- **migração**: pode ser usada para passar o *software* para um ambiente ou sistema diferente, menos caro, ou converter de uma linguagem de programação antiga e obsoleta para novas linguagens com mais recursos e flexibilidade;
- **conseguir maior confiabilidade**: este processo inclui atividades que revelam potenciais falhas, tornando, assim, o sistema mais estável e confiável;
- **preparação para adição funcional**: a decomposição do programa em módulos menores torna mais fácil a alteração ou adição de novas funcionalidades, sem afetar outros módulos.

Dentro deste tema existem várias terminologias utilizadas. Segue uma lista.

- **Engenharia reversa**: processo de analisar um tema para identificar seus componentes e interrelações, a fim de criar outra representação ou um nível mais elevado de abstração.

• ***Design recovery***: é um subconjunto da engenharia reversa onde conhecimentos do domínio e informações externas são acrescidas à observação para identificar abstrações de alto nível que estão além daquelas obtidas apenas pela observação do sistema.

• **Redocumentação**: é a criação ou revisão de uma semântica representativa equivalente com o mesmo nível de abstração. Produto são visões alternativas de fácil entendimento, como diagramas de fluxo de dados, estrutura e controle;

• **Reestruturação**: é a transformação de uma representação em outra no mesmo nível de abstração, preservando o comportamento externo do sistema.

• ***Forward engineering***: é o desenvolvimento de *software* tradicional que inicia com abstrações de alto nível e termina com implementações físicas.

• **Reengenharia**: é a análise e modificação de um sistema para reconstruí-lo de uma nova forma e sua posterior implementação.

Embora sejam termos semelhantes, existe uma diferença entre reengenharia e engenharia reversa. A reengenharia pode ser definida como um redesenho de processos no qual ocorre uma readequação dos processos empresariais, das estruturas organizacionais, dos sistemas de informação e dos valores da organização a fim de melhorar os resultados dos negócios da organização.
A engenharia reversa pode ser definida como uma forma de analisar o sistema com o objetivo de identificar os seus componentes e o seus interrelacionamentos ou também de criar outra representação do sistema ou em outros níveis mais altos de abstração.

A figura 2.4 mostra a relação entre estes termos e o ciclo de vida do *software* dividido em três estágios, requisitos: *design* e implementação.

A figura 2.5 mostra uma outra representação da reengenharia, porém compatível com a figura 2.4. Na figura 2.5, a entrada do processo é um sistema legado e a saída é uma versão estruturada e modular do mesmo sistema.
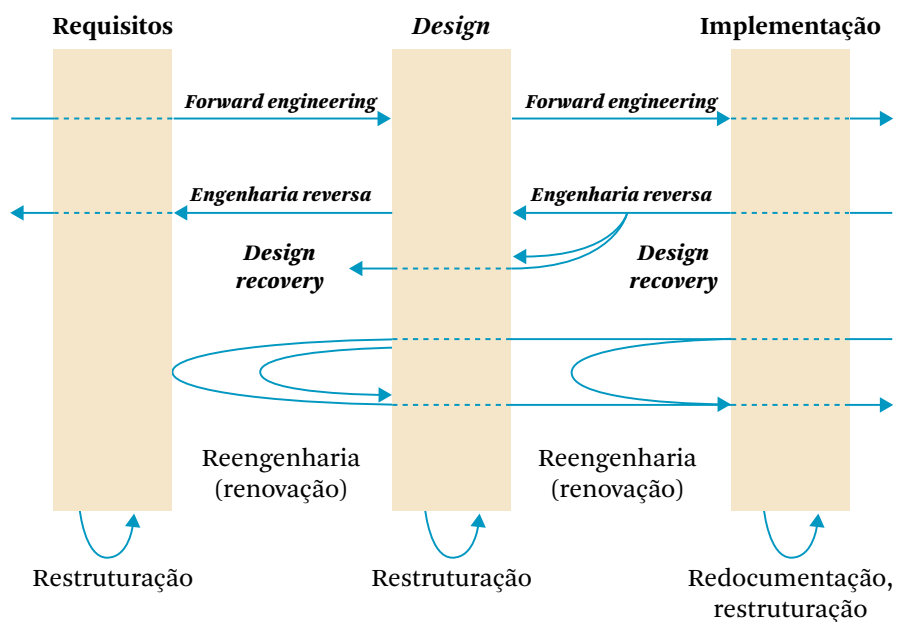
Figura 2.4 – Os 3 estágios da reengenharia. Fonte: SOMMERVILLE, 2007.
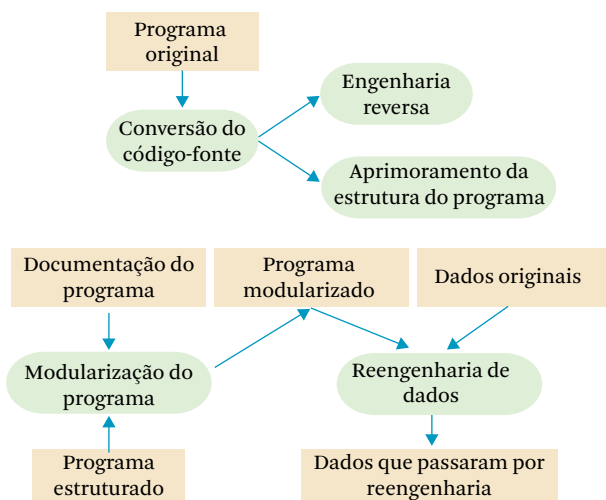
Figura 2.5 – Processo de reengenharia. Fonte: SOMMERVILLE, 2007.

Processo Evolucionário (Evolução de sistema)

A flexibilidade de sistemas de *software* é uma das principais razões pelas quais, cada vez mais, os *softwares* estão sendo incorporados em sistemas grandes e complexos. Uma vez tomada uma decisão de fabricar *hardware*, é muito dispendioso fazer modificações no projeto de *hardware*. Contudo, quando se trata de *software*, as mudanças podem ser feitas em qualquer momento, seja durante ou depois do desenvolvimento do sistema. Essas modificações podem ser muito onerosas, mas ainda são muito mais baratas do que as modificações correspondentes no sistema de *hardware*.

Historicamente, sempre houve um limite entre o processo de desenvolvimento de *software* e o processo ele evolução de *software* (manutenção de *software*). O desenvolvimento de *software* é considerado uma atividade criativa, em que um sistema de *software* é desenvolvido a partir de um conceito inicial até chegar ao sistema em operação. A manutenção de *software* é o processo de modificar esse sistema, depois que ele entrou em operação. Embora os custos de 'manutenção' sejam, com frequência, muitas vezes maiores do que os custos de desenvolvimento inicial, os processos de manutenção são considerados menos desafiadores do que o desenvolvimento de *software* original .

Esse limite está se tornando cada vez mais irrelevante. Poucos sistemas de *software* são, atualmente, sistemas completamente novos, e tem muito mais sentido considerar o desenvolvimento e a manutenção como estágios integrados e contínuos. Em vez de dois processos separados, é mais realista pensar na engenharia de *software* como um processo evolucionário, em que o *software* é continuamente modificado ao longo de seu tempo de duração, em resposta a requisitos em constante modificação e às necessidades do cliente. Esse processo evolucionário está ilustrado na figura 2.6 (SOMMERVILLE, 2006, p. 53).
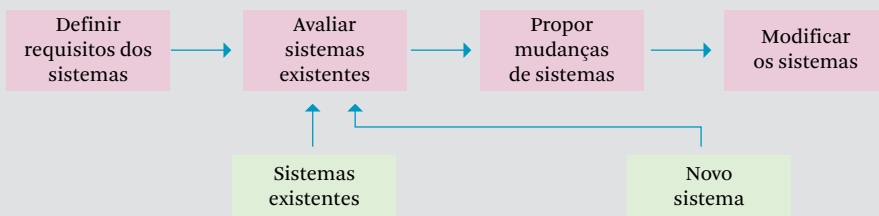
Figura 2.6 – Evolução de sistema. Fonte: SOMMERVILLE (2006, p.53) (Adaptado pelo autor)

2.4 Suporte e treinamento

Suporte pode ser definido, basicamente, como um "conjunto de atividades de engenharia que ocorrem depois que o *software* foi fornecido ao cliente e posto em operação", de acordo com Pressman (2011, p. 514), podendo ainda ser também chamado, em alguns casos, de *help desk* e *service desk*. O *help desk* está mais associado a serviços de atendimentos a clientes com o propósito de esclarecer dúvidas, dar informações e orientações acerca de eventuais problemas e tudo o que envolve o termo "ajuda". Já o termo *service desk* possui um envolvimento mais abrangente do que o *help desk*, considerado até mesmo uma evolução deste, já que no *service desk*, aspectos relacionados à qualidade e ao bom funcionamento dos sistemas são o foco principal, tendo uma forte relação com a manutenção, e se propõe à estabilidade dos serviços ou à sua retomada, de modo que minimize qualquer problema ou impactos decorrentes de eventuais problemas. Para isto, se faz necessário o uso de ferramentas que cuidem da Gestão de Serviços de TI de maneira estruturada.

Para que o negócio seja suportado adequadamente, é necessário que os usuários destes serviços de TI tenham soluções rápidas, e nem sempre isto ocorre, pois às vezes os usuários nem sabem ao certo a quem recorrer quando surgem eventuais problemas, ficando sem orientações quanto à uma solução eficaz e precisa (PEREIRA, 2010).

Em seu artigo, Pereira, (2010) cita uma importante afirmação de Magalhães e Pinheiro (2007), que dizem que "a Central de Serviços é a principal interface operacional entre a área de TI e os usuários dos seus serviços, ela permite a centralização da comunicação dos erros, dúvidas e solicitações relacionados com os serviços de TI". No mesmo artigo, o autor Pereira, (2010) também cita outra colocação realizada pelos autores Abreu e Fernandes (2008), afirmando que "esta função destina-se a responder de forma rápida e eficiente as questões, reclamações e problemas dos usuários, de forma a permitir que os serviços sejam executados com o grau de qualidade esperado".

É importante que a organização reflita qual seria a sua real contribuição da área de suporte para o seu negócio e que esta área saiba, principalmente, como contribuir. De acordo com Magalhães (2008), a partir desta reflexão poderíamos obter pontos de vistas distintos, como, por exemplo:

AGENTES	realização de um atendimento completo logo no primeiro contato do usuário.
ORGANIZAÇÃO	incremento da disponibilidade dos serviços de TI e redução do custo de atendimento.
TECNOLOGIA DA INFORMAÇÃO	maior produtividade da equipe de TI e melhora da imagem da área de TI.
CLIENTES E USUÁRIOS	maior disponibilidade e agilidade no atendimento

Uma das ferramentas de Gestão de Serviços muito utilizada nas organizações de TI é a ITIL (*Information Technology Infrastructure Library*). A ITIL e um conjunto de boas práticas para serem utilizadas principalmente nos serviços de TI e foi desenvolvida no final dos ano 1980 pela CCTA (*Central Computer and Telecomunications Agency*), conhecida atualmente por OGC (*Office for Government Commerce*), da Inglaterra, a partir de uma encomenda do governo britânico.

A ITIL possui diferentes versões, sendo que a versão de 2011 é uma atualização da Versão 3, lançada em 2007, e é composta por cinco publicações principais, a saber:

- Estratégia de Serviço
- Desenho de Serviço
- Transição de Serviço
- Operação de Serviço
- Melhoria Contínua de Serviço

Deste modo, o que é importante ressaltarmos aqui é a importância de processos estruturados para estes tipos de serviços que envolvem o suporte, dentro do contexto da TI. Usar ferramentas de Gestão de Serviços, onde possa ter padrões, métricas, indicadores e processos estruturados, auxilia as organizações a terem uma prestação de serviços de qualidade.

A gestão destes serviços de suporte dentro das organizações pode ser feita de diferentes formas, mas a necessidade de uma estrutura organizada, de processos definidos de relatórios de acompanhamento e painéis executivos que auxiliem os gestores, é imprescindível para que estas atividades sejam eficientes e que atendam adequadamente às necessidades dos clientes, usuários finais e de todos os envolvidos no processo da prestação deste serviço de suporte dentro da organização.

Para que o suporte seja eficaz, também é muito importante que a equipe prestadora deste serviço tenha acesso à documentação do sistema ou *software* em questão. Uma documentação completa e precisa reflete diretamente na qualidade do suporte, principalmente o suporte técnico que envolve rotinas de uma aplicação, funcionalidades específicas, bem como na compreensão de processos e regras desenvolvidas que poderão ser analisadas por meio de artefatos da documentação, como, por exemplo, diagramas, modelagens e outros.

O treinamento é um importante momento para todos os envolvidos em um processo de desenvolvimento de *softwares*. É o momento em que os usuários finais, aqueles para o qual o sistema ou *software* foi desenvolvido, irão aprender a utilizar o sistema e até mesmo validar alguns pontos que, teoricamente, já deveriam ter sido passados na fase de testes.

Um treinamento pode envolver pessoas com perfis funcionais diferentes no sistema, por isso é importante ter uma abordagem correta ao público presente para que não cause problemas de expectativas ou dificuldades durante este momento. Uso de guias e manuais do usuário são muito úteis para que sirva de orientação e apoio aos utilizadores do sistema. Mais uma vez notamos a importância de uma boa documentação, com seus diversos artefatos, que nesta hora do treinamento podem servir para extrair informações, telas, processos e roteiros que poderão compor estes manuais para o usuário.

As consequências de um bom e eficaz treinamento podem ser ótimas para uma organização, pois seus envolvidos poderão contribuir ainda mais para melhorias constantes e validação de requisitos importantes, uma vez que o seu envolvimento aumenta cada vez mais e este sente-se seguro, implicando

em aumento de produtividade e aproveitamento do sistema desenvolvido, podendo enxergar novas demandas futuras ou novas implementações que muito contribuirão para a melhoria contínua do *software*.

2.5 Melhoria contínua

Cada vez mais as empresas e organizações têm investido na melhoria de seus processos de desenvolvimento de *software*, em busca dos benefícios que esta ação proporciona, como o aumento da qualidade de seus produtos e diminuição dos esforços para produzi-los e mantê-los. No entanto, esta ação não é simples e requer uma série de esforços por parte dos gestores e não existe um método padronizado para a sua execução (HIRAMA e TAMAKI, 2007, p. 1).

As empresas têm percebido que um dos principais desafios é a gestão efetiva dos processos de desenvolvimento de *software*, em busca da garantia da qualidade de *software*. A qualidade, neste âmbito, pode ser considerada e vista sob três aspectos, sendo o primeiro associado aos requisitos de qualidade, o segundo, relacionado com qualidade do processo que produz o produto e o terceiro, que associa a qualidade do produto no contexto do ambiente em que ele será utilizado (COSTA, 2012, p. 7).

Definir processos é fundamental, mas apenas isto é insuficiente para a melhoria contínua. Os processos não ficam estáticos para sempre a partir destas definições e estes precisam passar por mudanças contínuas, refinando de acordo com as demandas que a organização gera ao longo de tempo, oferecendo o dinamismo necessário para que os processos acompanhem a evolução e complexidade do negócio, sendo então constantemente melhorados.

A introdução do termo melhoria de processo foi realizada por Humphrey (1989), onde foi estabelecido que processos de desenvolvimento de *software* capazes e adaptáveis são pré-requisitos fundamentais para alcançar altos níveis de qualidade em produtos de *software* (COSTA, 2012, p. 7).

O *framework* proposto por Humphrey (1989) era composto por níveis de maturidade de processos de *software* com o propósito de atingir a melhoria contínua, de modo que estes processos pudessem ser medidos e controlados. De acordo com Costa (2012, p. 7), este framework deu origem a iniciativas de melhorias de processos como o CCM/CMMI e o SPICE (*Software Process Improvement and Capability Determination*), que originou a ISO/IEC 15504

(ISO/IEC, 2003), proporcionando às organizações formas de atingir e alcançar melhorias de maneira incremental.

Ao longo dos anos, vários estudos foram realizados em prol da melhoria de processos de *software*, com o propósito de identificar as melhores práticas e os fatores críticos de sucesso para estas implantações e, com isso, tem-se observado cada vez mais o uso de modelos de maturidade e sistemas de qualidade nas empresas de desenvolvimento de *software*, conforme afirma Santos (2007).

Os conceitos da Gestão da Qualidade Total (TQM - *Total Quality Management*) foram incorporadas pela melhoria de processo de *software*, por meio do ciclo de melhoria contínua PDCA (*Plan, Do, Check, Action*).

O **Ciclo PDCA** é uma ferramenta de gestão com foco em melhora contínua muito utilizada por várias empresas em todo o mundo, também conhecido por Ciclo de Deming. De acordo com Pacheco (2006):

> O Ciclo PDCA, também conhecido como Ciclo de Shewhart, Ciclo da Qualidade ou Ciclo de Deming, é uma metodologia que tem como função básica o auxílio no diagnóstico, análise e prognóstico de problemas organizacionais, sendo extremamente útil para a solução de problemas. Poucos instrumentos se mostram tão efetivos para a busca do aperfeiçoamento quanto este método de melhoria contínua, tendo em vista que ele conduz a ações sistemáticas que agilizam a obtenção de melhores resultados com a finalidade de garantir a sobrevivência e o crescimento das organizações (QUINQUIOLO, 2002).

Este ciclo, demonstrado na figura 2.7, é baseado em um princípio de que a gestão de qualquer sistema de produção necessita de quatro atividades sequenciais, sendo: planejar ("Plan"), executar ("Do"), verificar ("Check") e agir ("Act"). De acordo com Chermont (2001), "o ciclo PDCA é uma maneira de se planejar efetivamente uma ação, sendo bastante adequado porque não impõe uma regra e sim uma filosofia de ação".
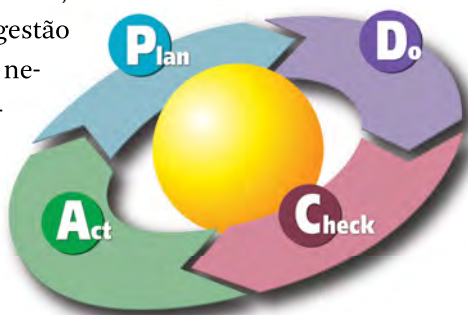
Figura 2.7 – O ciclo PDCA ou Ciclo de Deming. Disponível em: Wikimedia <http://commons.wikimedia.org/wiki/File:PDCA_Cycle.svg>

O estágio inicial do Ciclo PDCA é o planejamento da ação e, em seguida, é executado tudo o que fora planejado inicialmente, e ainda são realizadas verificações e novas ações. Por ser um ciclo, a cada vez os processos são melhorados, contribuindo para a melhora contínua.

Periard (2011) descreve detalhadamente cada etapa:

P = PLAN (PLANEJAMENTO)	Nesta etapa, o gestor deve estabelecer metas e/ou identificar os elementos causadores do problema e que impedem o alcance das metas esperadas. É preciso analisar os fatores que influenciam este problema, bem como identificar as suas possíveis causas. Ao final, o gestor precisa definir um plano de ação eficiente.
D = DO (FAZER, EXECUÇÃO)	Aqui é preciso realizar todas as atividades que foram previstas e planejadas dentro do plano de ação.
C = CHECK (CHECAGEM, VERIFICAÇÃO)	Após planejar e por em prática, o gestor precisa monitorar e avaliar constantemente os resultados obtidos com a execução das atividades. Avaliar processos e resultados, confrontando-os com o planejado, com objetivos, especificações e estado desejado, consolidando as informações, eventualmente confeccionando relatórios específicos.
A = ACT (AÇÃO)	Nesta etapa é preciso tomar as providências estipuladas nas avaliações e relatórios sobre os processos. Se necessário, o gestor deve traçar novos planos de ação para melhoria da qualidade do procedimento, visando sempre a correção máxima de falhas e o aprimoramento dos processos da empresa.

Vale ressaltar que este ciclo pode seguir constantemente, sem ter a necessidade de ter um fim determinado. O importante é efetuar as ações corretivas quando terminar o primeiro ciclo e efetuar o planejamento do próximo ciclo já realizando estes ajustes, promovendo a mehoria propriamente dita a cada ciclo. Caso uma etapa não seja respeitada ou não executada, o ciclo pode ser comprometido de modo significativo, por isso estes processos devem ser conduzidos de forma contínua, fazendo jus ao princípio deste método (PERIARD, 2011).

Existem diversos padrões, referências e modelos reconhecidos e utilizados em empresas e organizações que visam à melhoria da qualidade dos processos de desenvolvimento de *software*, entre eles destacam-se o ISO/IEC 15504, o CMMI (*Capability Maturity Model Integration*), o PMBoK (*Project Management Body of Knowledge*) e o SWEBoK (*Software Engineering Body of Knowledge*) (HIRAMA e TAMAKI, 2007, p. 1), além das normas ISO-9000:2000, a ISO/IEC 12207 e o MR-MPS.

Veremos algumas destas normas e modelos no próximo capítulo.

ATIVIDADES

01. Cite as principais características da documentação ágil.

02. Durante o processo de desenvolvimento de *software*, o que se pode documentar? Cite os possíveis artefatos que podem ser gerados e comente.

03. Em sua opinião, é possível efetuar manutenção de um *software* totalmente desprovido de materiais ou artefatos de documentação? Explique.

04. Quando o desenvolvimento de um *software* é entregue ao cliente, pode-se considerar o fim do seu ciclo de vida? Por quê?

05. O suporte e/ou o treinamento devem-se iniciar apenas após a entrega final do *software*, ou estes poderiam ocorrer durante o desenvolvimento do mesmo, em entregas parciais?

06. Até quando o ciclo de melhoria contínua de um *software* deveria ser utilizado? Explique.

REFLEXÃO

Um *software* ou sistema acaba quando ele é entregue e colocado em produção?

Vimos que o *software* ou sistema ainda demandam muitas atividades após a sua entrega final ao cliente ou depois que é colocado em produção. Se considerarmos apenas o projeto, que havia escopo, plano, prazo e custo definidos, talvez podemos considerar a entrega final como o seu fim, do projeto. Mas para o *software* ainda não, pois o seu ciclo de vida ainda se estende com o seu uso e com as fases de manutenção, que podem assumir um papel de evolução deste *software*, além das ações corretivas e preventivas. Desta forma, um *software* ou sistema não acaba ou termina quando ele é entregue e colocado em produção. Muitas vezes, isto pode ser apenas o começo de uma longa trajetória que este sistema pode percorrer ao longo de sua vida útil, com evoluções e novas versões que cada vez são incrementadas com mais funcionalidades ou recursos, melhorando-o cada vez mais. Também pudemos notar que muitas vezes o custo envolvido no processo de manutenção do *software* pode ser bem superior ao seu custo de projeto, e o tempo de uso evidentemente é muito maior que o período no qual ele está sendo desenvolvido.

Assim, vemos a importância da documentação do *software*, que garantirá condições de efetuar uma manutenção técnica mais eficaz, além de trazer maior segurança para o suporte técnico, que poderá intervir e auxiliar de maneira mais precisa.

A melhoria contínua promove a garantia de evolução do *software*, que como o próprio nome sugere, sempre com melhorias. A renovação de alguns processos e suas revisões provocam um estado constante de aperfeiçoamento, que vai garantindo a qualidade do *software*, atrelado a normas e padrões reconhecidos e suportados por órgãos e institutos especializados.

Tente imaginar como seria um *software* totalmente desprovido de documentação técnica e sem qualquer tipo de manutenção após seu uso. Certamente seria um caos e muito em breve seus requisitos poderão não atender ou não se adequar às regras de negócio do qual ele está inserido, e isto certamente será um problema que poderá causar o abandono e a diminuição no seu ciclo de vida.

Lembre-se, em relação à manutenção: você poder gastar pouco agora ou gastar muito mais tarde!

LEITURA

Embora a manutenção e o suporte de *software* sejam as atividades que possuem maior custo na vida útil de sistemas, estes são os assuntos que menos possuem materiais e livros do que qualquer outro tópico de assuntos relacionados à Engenharia de *Software*.

De qualquer maneira, podemos mencionar importantes obras recentes da literatura, em inglês, de acordo com Pressman, 2011:

- *Effective Software Maintenance and Evolution* (de Auerbach Jarzabeck, 2007).
- *Software Maintenance: Concepts and Practice, World Scientific Publishing Co., 2d. ed., 2003,* de Grubb e Takang.
- *Practical Software Maintenance* (Wiley, 1996).

REFERÊNCIAS BIBLIOGRÁFICAS

ABREU, V. F. De.; FERNANDES, A. A.; **Implantando a Governança de TI: uma estratégia à gestão de processos e serviços**. 2. ed. Rio de Janeiro: Brasport, 2008. Acesso em: 15 jan. 2015.

AMBLER, S. W.; **Modelagem Ágil - Práticas eficazes para a Programação Extrema e o Processo Unificado**. Bookman. 2004. p. 351.

COELHO, H. S.; **Documentação de Software: uma necessidade**. Texto Livre; nº 2; vol. 1; UFMG; 2009.

COSTA, T. M., **Melhoria Contínua de Processo de Software utilizando a teoria das restrições**. Dissertação de Mestrado UFRJ. Rio de Janeiro, 2012. p. 233.

FAGUNDES, R. M., **Engenharia de Requisitos - Do perfil do analista de requisitos ao desenvolvimento de requisitos com UML e RUP**. Salvador, 2011. p. 216.

HIRAMA, T.; TAMAKI, P. A. O.; 2007. **Melhoria de Processos de Desenvolvimento de Software Aplicando Process Patterns**. Disponível em: <http://www.dcc.ufla.br/infocomp/artigos/v6.1/art09.pdf> Acesso em: 04 dez. 2014.

HUMPHREY, W.S., 1989, **Managing the Software Process,** Boston, Addison-Wesley Professional.

MAGALHÃES, I. L. R. G.; **Indicadores de Desempenho para Service Desk - baseado na Strategic Activity System**. Apresentação. 2008. Disponível em: <http://pt.slideshare.net/ivan_luizio/indicadores-de-desempenho-para-service-desk-com-base-na-sas-presentation>. Acesso em: 14 fev. 2015.

MAGALHÃES, Ivan Luizio; PINHEIRO, Walfrido Brito. **Gerenciamento de serviços de TI na prática: uma abordagem com base na ITIL**. São Paulo: Novatec Editora, 2007. Série gerenciamento de TI.

MAZZOLA, V. B. **Engenharia de Software - Conceitos Básicos**, Apostila, 2010.

MICHELAZZO, P. **A Documentação de software**, 2006. Disponível em: <http://www.michelazzo.com.br/node/123>. Acesso em: nov. 2008.

PACHECO, A. P. R; SALLES, B. W.; GARCIA, M. A.; POSSAMAI, O. **O Ciclo PDCA na gestão do conhecimento: uma abordagem sistêmica**. Santa Catarina. 2006.

PEREIRA, U. S.; **Suporte Técnico de TI Baseado na Metodologia ITIL: um estudo de caso sobre o nível de maturidade em uma central de serviços**. Monografia. UnB. 2010. 46p. Disponível em: <http://bdm.unb.br/bitstream/10483/2751/1/2010_UlissesSepulvedoPereira.pdf>. Acesso em: 03 fev. 2015.

PERIARD, G. **O Ciclo PDCA e a melhoria contínua**. 2011. Disponível em: <http://www.sobreadministracao.com/o-ciclo-pdca-deming-e-a-melhoria-continua/>.

QUINQUIOLO, J. M. **Avaliação da Eficácia de um Sistema de Gerenciamento para Melhorias Implantado na Área de Carroceria de uma Linha de Produção Automotiva.** Taubaté-SP: Universidade de Taubaté, 2002.

SANTOS, G., MONTONI, M., VASCONCELLOS, J., et al., "**Implementing software process improvement initiatives in small and medium-size enterprises in Brazil**", 2007.

SAUVÉ, J. P.; **Processos de Desenvolvimento de Software**. 2001. Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/proj/gerenciadesenv/processos.htm>. Acesso em: 14 dez. 2014.

SOMMERVILLE, I. **Engenharia de Software**, 6. ed. Editora Pearson do Brasil, 2003. p. 292.

SOMMERVILLE, I. **Engenharia de Software**, 7. ed. Editora Pearson do Brasil, 2007.

TRIPATHY, P; NAIK, K; **Software Evolution and Maintenance**. New Jersey: Willey. 2015. p. 393.

WAZLAWICK, R. S. **Engenharia de Software: Conceitos e Práticas**, 1. ed., Elsevier, 2013. p. 506.

3

Introdução aos Padrões de PDS

Este capítulo abordará alguns dos padrões utilizados na Engenharia de *Software* para os Processos de Desenvolvimento do *Software* (PDS), relacionando-os com a qualidade e a importância destes processos.

Algumas entidades e organizações internacionais têm como objetivo a padronização de normas, testes e certificações entre os países, com o intuito de proporcionar maior qualidade ao produto de *software* e estas atuam com diferentes enfoques, com diferentes normas e padrões, respectivamente. A ISO – International Organization for Standardization, um organismo das Nações Unidas, a SEI – *Software* Engineering Institute e a ABNT – Associação Brasileira de Normas Técnicas são alguns destes exemplos.

Além de normas que visam essencialmente à qualidade do *software*, existem modelos que indicam práticas específicas e genéricas para obter um nível de maturidade (ou capacidade) de seus processos, que também estão relacionados diretamente à qualidade. De acordo com (SILVA, 2014), a utilização de modelos de maturidade tem sido realizada principalmente por organizações de grande porte, devido aos custos de implantação e também pelo objetivo de atingir um nível de maturidade certificado.

Veremos dois dos principais modelos mais utilizados no Brasil para a melhoria dos processos de *software*, o CMMI e MPS.BR.

OBJETIVOS

- Aprender sobre alguns dos principais padrões de PDS (Processos de Desenvolvimento de *Software*);
- Conhecer o modelo da capacidade de maturidade (CMM);
- Aprender sobre a norma ISO 15504, conhecida por SPICE;
- Conhecer a norma ISO 12207, que trata dos processos do ciclo de vida do *software*;
- Conhecer o modelo de qualidade de processo MPS/BR.

3.1 Introdução

Sabemos que existem processos bem definidos para desenvolver um produto de *software*, mas será que todas as organizações os seguem de uma mesma maneira ou será que cada uma adota seus processos a seu critério. Sim, é possível as empresas adotarem seus próprios processos ou adaptar alguns já conhecidos. Mas imagine se existissem alguns padrões já bem estruturados, por meio de normas que visassem a qualidade ou até mesmo alguns modelos de maturidade, do ponto de vista de processos. Ao adotar padrões de desenvolvimento e ao definir os processos em uma empresa, esta está ganhando em maturidade e contribuindo para a qualidade do produto final, devido à prática de processos bem estabelecidos.

Desta forma, existem diversos padrões de desenvolvimento de *software*, onde cada um possui um enfoque específico que busca normatizar ou trazer modelos a serem seguidos pelas organizações para estabelecer as melhoras práticas em relação aos seus processos e a fatores críticos de sucesso.

A maturidade de uma empresa é avaliada pela definição e estabelecimento dos processos nela praticados. Alguns modelos classificam uma empresa em diferentes níveis de maturidade, trazendo áreas-chaves para cada nível, auxiliando a organização a focar seus esforços em áreas determinantes para o seu momento e conseguir se estruturar de acordo com seus processos. Quanto mais bem definido e estabelecido for os processos em uma empresa, maior será a sua maturidade. Por isso é importante não confundir maturidade com tempo cronológico. Uma empresa relativamente nova, em termos de idade de vida, pode ser muito mais madura do que uma outra empresa que tenha décadas de vida, pois a primeira pode ter seus processos definidos e bem estruturados, diferentemente da segunda que, eventualmente, pode não ter processos tão bem estruturados, mesmo com um tempo maior de existência. Evidentemente que a maturidade também não está associada à idade das pessoas que ali trabalham e sim ao grau de maturidade de seus processos dentro da organização.

Vamos ver alguns modelos de maturidade e padrões que trabalham essencialmente com processos.

3.2 CMM / CMMI

3.2.1 CMM (Capability Maturity Model)

Se você já ouviu falar sobre CMM como uma metodologia de desenvolvimento ou qualidade de *software* infelizmente você ouviu errado. O CMM, como o nome diz, é um modelo de maturidade e sua tradução expressa exatamente a sua finalidade: modelo de maturidade da capacidade.

O CMM foi concebido pelo *Software* Engineering Institute, um departamento da Carnegie Mellon *University* a pedido do Departamento de Defesa dos EUA. Como já vimos em outras unidades, é uma demanda crescente do mercado a existência de *softwares* com mais qualidade e critérios.

O Departamento de Defesa dos EUA, com o objetivo de avaliar os seus fornecedores de *software* pediu ao SEI que desenvolvesse um modelo para medir a qualidade dos *softwares* que estavam sendo usados e desenvolvidos dentro do departamento.

O CMM serve para avaliar e melhorar a capacitação de empresas que produzem *software*. Embora o modelo CMM não tenha relacionamento com as normas da ISO, o modelo tem tido uma grande aceitação mundial, até mesmo fora do mercado americano. No Brasil existem várias empresas que usam este modelo.

O modelo, publicado em 1992, não é extenso e pode ser obtido na própria Internet com facilidade, porém possui um livro impresso com todos os detalhes (*Software* ENGINEERING INSTITUTE, 1997). Atualmente, o CMM também é chamado de SW-CMM (*Software* CMM).

Assim como outros modelos relacionados com qualidade, o CMM também foi baseado em algumas das ideias mais importantes na área de qualidade industrial. Estes conceitos foram adaptados para a área de *software* por Watts Humphrey, um dos idealizadores do CMM. A primeira versão oficial do CMM foi divulgada no final dos anos 80 e é descrita em relatórios técnicos do SEI em um livro, como já citado.

Mesmo com base na qualidade industrial, o CMM é específico para a área de *software*. Áreas importantes para a sobrevivência de uma organização produtora de *software* como *marketing*, finanças, administração não estão incluídos no CMM. Mesmo áreas importantes da informática como *hardware* e bancos de dados estão fora do escopo do CMM.

Logo, a aplicação do CMM, ou modelos equivalentes, não garante por si só a viabilidade de uma organização. O CMM deve estar integrado a outras iniciativas da organização da área de qualidade e assim, junto com estas outras áreas, torna-se um fator importante de melhoria da eficácia e competitividade.

3.2.1.1 Maturidade

O CMM é um modelo para medição da maturidade de uma organização no que diz respeito ao processo de desenvolvimento de *software*. Na tabela 3.1 apresentamos um comparativo entre organizações imaturas e maduras para podermos entender um pouco sobre maturidade em *software*.

ORGANIZAÇÕES MADURAS	ORGANIZAÇÕES IMATURAS
Papéis e responsabilidades bem definidos.	Processo improvisado.
Existe base histórica.	Não existe base histórica.
É possível julgar a qualidade do produto.	Não há maneira objetiva de julgar a qualidade do produto.
A qualidade dos produtos e processos é monitorada.	Qualidade e funcionalidade do produto sacrificadas.
O processo pode ser atualizado.	Não há rigor no processo a ser seguido.
Existe comunicação entre o gerente e seu grupo.	Resolução de crises imediatas.

Tabela 3.1 – Comparativo entre organizações maduras e imaturas.

3.2.1.2 Níveis

Existem 5 níveis de maturidade usados no CMM. Cada um dos níveis possui características próprias.

Segundo o CMM (1997), o nível 1 compreende a maioria das organizações, é chamado de *Inicial* e engloba as organizações mais imaturas. Neste nível não há nenhuma metodologia implementada e tudo ocorre de forma desorganizada.

No nível 5, chamado de *otimizado*, onde estão as organizações mais maduras, cada detalhe do processo de desenvolvimento está definido, quantificado e acompanhado e a organização consegue até absorver mudanças no processo sem prejudicar o desenvolvimento.

A tabela 3.2 mostra os níveis e uma breve descrição.

NÍVEL CMM	DESCRIÇÃO
1 – INICIAL	O processo de desenvolvimento é desorganizado e até caótico. Poucos processos são definidos e o sucesso depende de esforços individuais e heroicos.
2 – REPETÍVEL	Os processos básicos de gerenciamento de projeto estão estabelecidos e permitem acompanhar custo, cronograma e funcionalidade. É possível repetir o sucesso de um processo utilizado anteriormente em outros projetos similares.
3 – DEFINIDO	Tanto as atividades de gerenciamento quanto de engenharia do processo de desenvolvimento de *software* estão documentadas, padronizadas e integradas em um padrão de desenvolvimento da organização. Todos os projetos utilizam uma versão aprovada e adaptada do processo padrão de desenvolvimento de *software* da organização.
4 – GERENCIADO	São coletadas medidas detalhadas da qualidade do produto e processo de desenvolvimento de *software*. Tanto o produto quanto o processo de desenvolvimento de *software* são entendidos e controlados quantitativamente.
5 – OTIMIZADO	O melhoramento contínuo do processo é conseguido através de um *feed-back* quantitativo dos processos e pelo uso pioneiro de ideais e tecnologias inovadoras.

Tabela 3.2 – Os níveis do CMM.

3.2.1.3 Áreas-chaves – KPA

Cada nível do CMM (exceto o nível 1) é composto de várias áreas-chaves de processo (*key process areas*, ou KPAs). Cada área-chave identifica um grupo de atividades correlatas que realizam um conjunto de metas consideradas importantes, quando executadas em conjunto. As áreas-chaves de um nível identificam as questões que devem ser resolvidas para atingir este nível.

No CMM, cada área-chave reside em um único nível de maturidade. Para ser classificada em um determinado nível, uma organização tem de ter implementado completamente as áreas-chaves deste nível e todos os níveis inferiores.

Cada área-chave define um conjunto de metas, que representam o estado atingido por uma organização que domine a área-chave. Para atingir as metas da área-chave, a organização deve implementar um conjunto de práticas-chaves.

Estas práticas descrevem procedimentos gerenciais e técnicos, descritos no CMM, com grau de detalhe variável. A figura 3.1 expressa essas práticas-chave.
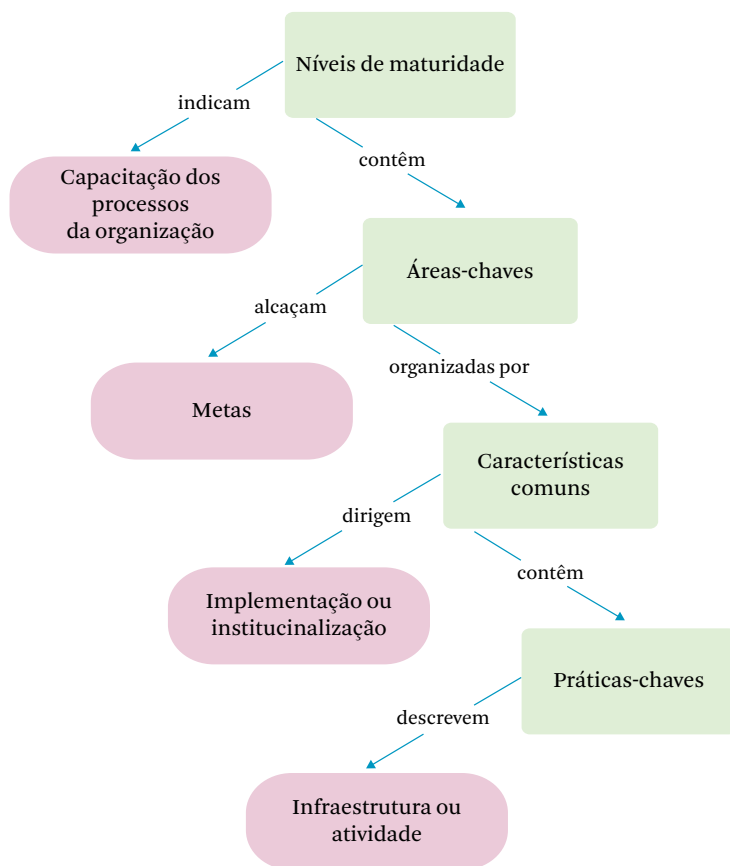
Figura 3.1 – Áreas-chave do CMM.

Um ponto interessante no CMM é o compromisso com a estabilidade da melhoria já conseguida. Algumas empresas tendem a voltar a praticar os velhos e ruins hábitos de processo errado após a obtenção da certificação. Afinal, trabalhar com maturidade e *software* não é uma tarefa tão fácil.

Para isto, cada área-chave possui vários grupos de atividades de institucionalização. Essas atividades, quando executadas, forçam a empresa a continuar fazendo o que foi feito anteriormente para conseguir a melhoria, dificultando, assim, o retrocesso nas atividades de implementação. As atividades de institucionalização são divididas nos seguintes grupos, chamados de Características Comuns (*Common Features*):

- **Comprometimento em executar (*Commitment to Perform*)**: descrevem as ações que a organização deve ter para garantir que o processo foi implantado e qual ele persiste. Estas ações envolvem ter uma política organizacional documentada e boa liderança.
- **Capacitação para executar (*Ability to Perform*)**: descreve as pré-condições que devem existir no projeto ou na organização para implementar o processo de *software* com competência. Envolvem recursos, estrutura organizacional e treinamento.
- **Atividades realizadas (*Activities Performed*)**: descreve as atividades, papéis e procedimentos necessários para implementar uma KPA. Envolvem a implantação de planejamento e procedimentos, realização do trabalho, acompanhamento e tomar ações corretivas, se necessário.
- **Medição e análise (*Measurement and Analysis*)**: medições básicas necessárias para avaliar o *status* da área-chave. Estas medidas são usadas para controlar e melhorar o processo.
- **Verificação da implementação (*Verifying Implementatton*)**: ações que garantem a conformidade das demais atividades com os processos estabelecidos. Envolvem as ações de verificação por parte da gerência superior da organização, dos gerentes dos projetos e de um grupo independente de garantia da qualidade.

Apresenta-se como exemplo, a seguir, a estrutura da área-chave de gestão de requisitos. Esta área é a primeira do nível do CMM, e uma das mais simples em quantidade de práticas-chaves.

A documentação do CMM apresenta maiores detalhes a respeito de cada prática. A descrição apresentada na tabela 3.3 – Exemplo da KPA de Gestão de Requisitos – é simplificada em relação à definição oficial das práticas do CMM.

<table>
<tr><td>METAS</td><td colspan="2">Atividades de engenharia e gestão do software baseadas em requisitos documentados.
Consistência permanente de planos, produtos e atividades com os requisitos.</td></tr>
<tr><td rowspan="5">PRÁTICAS-CHAVE</td><td>Comprometimento em executar</td><td>Existência de política escrita para Gestão de requisitos.</td></tr>
<tr><td>Capacitação para executar</td><td>Designação de responsáveis pelos requisitos, em todos os projetos.
Documentação dos requisitos alocados ao software.
Existência de recursos e orçamento adequados para Gestão de Requisitos.
Treinamento da equipe de software e equipes correlatas em Gestão de requisitos.</td></tr>
<tr><td>Atividades a executar</td><td>Revisão prévia dos requisitos pelos grupos afetados.
Uso dos requisitos como base para planos, produtos e atividades.
Revisão e incorporação ao projeto das mudanças de requisitos.</td></tr>
<tr><td>Medição e análise</td><td>Status das atividades de Gestão de requisitos.</td></tr>
<tr><td>Verificação da implementação</td><td>Revisão periódica das atividades de Gestão de requisitos pela gerência executiva.
Revisão periódica e por eventos das atividades de Gestão de requisitos pelos gerentes dos projetos.
Revisão e auditoria das atividades de Gestão de requisitos pelo Grupo de garantia da qualidade de software.</td></tr>
</table>

Tabela 3.3 – Exemplo da KPA de Gestão de Requisitos.

As metas a atingir requerem que todas as atividades técnicas e gerenciais dos projetos sejam baseadas nos requisitos do produto que será desenvolvido, e que os artefatos produzidos sejam todos consistentes com estes requisitos. Para isto, prevêem-se três atividades a executar:

- os requisitos devem sofrer revisão prévia dos grupos afetados, dos quais o grupo dos desenvolvedores é o mais óbvio;
- estes requisitos devem ser usados como base para o planejamento e para as diversas atividades do projeto;
- se os requisitos forem alterados ao longo de um projeto, estas alterações devem ser aprovadas pelos grupos afetados e incorporadas de forma disciplinada.

Para garantir que estas atividades sejam executadas de forma permanente e estável, o CMM requer que a organização tenha uma política documentada para a área; não basta existir um costume não escrito.

Todo projeto tem de ter um responsável pelos requisitos, oficialmente designado. Todos os requisitos alocados ao *software* (dentre os requisitos do sistema) têm de ser documentados. As atividades de gestão de requisitos devem receber recursos suficientes e o pessoal envolvido deve ser treinado no assunto.

Devem ser feitas medições de *status* da gestão de requisitos, por exemplo, o grau de estabilidade dos requisitos de cada projeto. E, finalmente, a Gestão de requisitos deve ser acompanhada, em nível mais alto, pela alta direção da organização, chamada no CMM de Gerência Executiva (*sênior management*) em nível detalhado, pelos gerentes dos respectivos projetos e por um grupo independente de Garantia da qualidade, que forneça à gerência executiva informações independentes dos gerentes de projeto.

Com pequenas variações, as atividades de institucionalização das outras áreas-chaves do CMM funcionam de forma semelhante. Por outro lado, as atividades de implementação são bastante específicas de cada área-chave.

3.2.1.4 Nível 1 – Inicial

A organização nível 1 representa o estágio inicial dos produtores de *software*. Ela utiliza processos informais e métodos *ad hoc*, às vezes descritos como caóticos. Muitas destas organizações são bem sucedidas, já que o mercado de *software* é ainda tolerante em relação à má qualidade dos produtos. Muitas vezes, a qualidade do *marketing* pode ocultar deficiências técnicas, ou existe pouca competição em muitos setores deste mercado. A cultura no nível inicial é muito baseada no valor dos indivíduos. É comum a dependência em relação a heróis técnicos e gerenciais.

A organização nível 1 geralmente não é capaz de fazer estimativas de custo ou planos de projeto e se faz, não é capaz de cumpri-los. As ferramentas não são integradas com os processos e não são aplicadas com uniformidade pelos projetos. Geralmente, a codificação é a única fase dos processos de desenvolvimento que merece atenção. Engenharia de requisitos e desenho são fracos ou inexistentes, e mudanças de requisitos e de outros artefatos ocorrem sem controle. A instalação e manutenção costumam ser deficientes, sendo encaradas como atividades de pouca importância.

Em geral, os gerentes destas organizações não entendem os verdadeiros problemas devido a uma falta de processos nos quais possam apresentar a real situação do projeto em relação ao seu progresso. Como já foi citado, normalmente são pessoas que entendem da parte administrativa, mas não entendem os problemas de *software*, ou vice-versa, e o gerente é um excelente técnico que foi subindo na carreira, mas lhe falta a parte administrativa.

Podem ocorrer também que existam processos definidos no papel e não são aplicados realmente, ou são sempre contornados, com o apoio e até a pressão dos gerentes. Na melhor das hipóteses, os processos são seguidos quando os projetos estão em fase tranquila; em crise, abandonam-se os métodos e reverte-se à codificação sem critérios.

3.2.1.5 Nível 2 – Repetível

Como o nome diz, o tema das organizações nível 2 é serem capazes de cumprir compromissos e repetir o que já foi acertado. No nível repetível, uma organização é capaz de assumir compromissos referentes a requisitos, prazos e custos com alta probabilidade de ser capaz de cumpri-los. Isso já é um grande começo na carreira de maturidade em *software*!

Porém, requer o domínio das seguintes áreas-chaves.

- ***RM – Requirements Management***: a Gestão dos requisitos permite definição e controle dos requisitos em que se baseiam os compromissos acordados entre cliente e desenvolvedor;
- ***SRP – Software Project Planning***: o Planejamento de projetos prevê prazos e custos para cumprimento dos compromissos, como bases técnicas e não apenas intuitivas;

- ***SPTO – Software Project Tracking and Oversight***: a Supervisão e Acompanhamento de projetos confere o atendimento dos compromissos, comparando o conseguido com o planejamento e acionando providências corretivas sempre que haja desvios significativos em relação aos compromissos;
- ***SSM – Software Subcontract Management***: a Gestão da subcontratação cobra de organizações subcontratadas para desenvolver partes do *software* os mesmos padrões de qualidade que a organização principal oferece a seus clientes;
- ***SQA – Software Quality Assurance***: esta KPA estabelece um grupo de Garantia da qualidade de *software* que confere o cumprimento dos compromissos, de forma independente em relação aos projetos;
- ***SCM – Software Configuration Management***: a Gestão de configuração garante a consistência permanente dos resultados dos projetos, entre si e com os requisitos, ao longo do projeto, mesmo quando ocorram alterações nos compromissos.

No nível 2, as organizações são disciplinadas em nível dos projetos. Por isto, elas conseguem estimar e controlar projetos iguais aos quais já foram anteriormente bem-sucedidos. Porém, estas organizações correm os riscos diante de vários tipos de mudanças a que as organizações estão sujeitas, tais como:

- mudanças de ferramentas e métodos, conforme ocorre a evolução de tecnologia;
- mudanças de estrutura organizacional, causadas por diversos fatores da dinâmica das organizações.
- mudanças de tipos de produto, causadas por variações dos mercados.

3.2.1.6 Nível 3 – Definido

As organizações do nível 3 são aquelas que passam da gestão de projetos simplesmente para a engenharia de produtos. É óbvio que as práticas do nível 2 devem estar implantadas e em prática.

No nível 3, o sucesso conseguido em projetos anteriores deve ser repetido e, além disso, uma infraestrutura de processos que permitiu a adaptação a vários tipos de mudanças também deve ser estabelecida.

As áreas-chave deste nível são:

- ***Organization Process Definition*** **– Definição do Processo da Organização:** consiste em estabelecer formalmente na organização um grupo de processos de engenharia de *software*, responsável pelas atividades de desenvolvimento, melhoria e manutenção de processos de *software*;
- ***Organization Process Focus*** **– Foco nos Processos da Organização**: consiste em estabelecer um processo-padrão de *software* a nível da organização que seja um modelo para outros projetos e contenha os mesmos processos já definidos;
- ***Training Program*** **– Programa de Treinamento**: consiste em estabelecer um programa de treinamento em processos de *software*, no nível da organização para os envolvidos nos projetos;
- ***Integrated Software Management*** **– Gestão Integrada de *Software***: consiste em integrar as atividades de engenharia de *software* e atividades gerenciais em um processo definido de *software* que provém do processo-padrão definido na KPA "Definição do Processo da Organização" com o uso de procedimentos documentados para gestão de tamanho, esforços, prazos e riscos;
- ***Software Product Engineering*** **– Engenharia de Produto de *Software***: consiste em definir a padronização a nível da organização dos métodos de engenharia de produtos de *software*, abrangendo engenharia de requisitos, testes, desenho, codificação e documentação de uso;
- ***Peer Reviews*** **– Revisão em Pares**: consiste em desenvolver atividades a fim de remover defeitos dos produtos de trabalho de *software* o mais cedo possível e com eficiência. Os pares citados na definição são os envolvidos no projeto;
- ***Intergroup Coordination*** **– Coordenação Entre Grupos**: estabelece atividades de coordenação de revisões técnicas a nível da organização.

A organização nível 3 sabe manter-se dentro do processo, mesmo durante as crises. As ferramentas passam a ser aplicadas de forma sistemática, padronizada e coerente com os processos. Com isto, o manual do engenheiro de *software* passa a contribuir significativamente para melhoria da produtividade e qualidade.

Por outro lado, o conhecimento dos processos, por parte da organização nível 3, ainda é basicamente qualitativo. Existe uma base de dados de processos, povoada com os dados recolhidos dos projetos. Esta base é usada para gestão dos projetos, mas não é ainda aplicada, de forma sistemática e no nível da organização, para atingir metas quantitativas de desempenho de processo e de qualidade de produto.

3.2.1.7 Nível 4 – Gerenciado

Na organização nível 4, o domínio dos processos de *software* evolui para uma forma quantitativa. Isso não quer dizer que apenas organizações deste nível devam coletar métricas de processo.

Todas as áreas-chaves do CMM contêm pelo menos uma prática de medição e análise, que sugere métricas adequadas para medir o sucesso da implantação da respectiva área. A organização nível 3 constrói e mantém uma base de dados de processos.

Coletar dados não é uma tarefa fácil e, além disso, é uma atividade cara, pois é necessário definir com precisão e antecipadamente quais os dados que serão coletados.

Estes dados precisam ser criticados, consistidos e normalizados para terem a qualidade desejada

No nível 4, a organização aprende e executa a coleta de métricas e administra a base de dados de processo, que é analisada e administrada por profissionais treinados. Além disso, os profissionais sabem intervir nos processos para atingir metas de qualidade dos produtos.

Este nível tem apenas duas áreas-chaves:

***QUANTITATIVE PROCESS MANAGEMENT* – GESTÃO QUANTITATIVA DO PROCESSO**	consiste em possuir atividades relacionadas com a gestão quantitativa dos processos e controle do desempenho dos processos usados pelos projetos;
SOFTWARE QUALITY MANAGEMENT* - GESTÃO DA QUALIDADE DE *SOFTWARE	consiste em promover o entendimento quantitativo da qualidade dos produtos de *software*, permitindo atingir metas quantitativas desejadas.

A organização que está no nível 4 passa da engenharia de produtos (do nível 3) à qualidade de processos e produtos.

Ela passa a ter elementos para decidir, por exemplo, e com base em valores quantitativos, qual deve ser a fração de recursos dos projetos destinada à garantia da qualidade, considerando o nível máximo de defeitos que se deseja nos produtos. Estas informações devem estar na base de dados citada.

Com o domínio quantitativo dos processos conseguido, é possível chegar a estados de melhoria contínua.

3.2.1.8 Nível 5 – Otimizado

De acordo com o CMM, quando a organização está no nível 5 ela atinge um estado em que os processos estão em melhoria contínua, sendo otimizados para as necessidades de cada momento. O nome do nível resume a meta desta fase.

O nível 5 possui as seguintes áreas-chave:

DEFECT PREVENTION **– PREVENÇÃO DE DEFEITOS**	consiste em desenvolver atividade para a prevenção dos defeitos por meio da identificação e remoção das suas causas
TECHNOLOGY CHANGE MANAGEMENT **– GESTÃO DA EVOLUÇÃO TECNOLÓGICA**	consiste em prover atividades para a gestão da evolução tecnológica, com procedimentos sistemáticos de identificação, análise e introdução de tecnologia apropriada;
PROCESS CHANGE MANAGEMENT **– GESTÃO DA EVOLUÇÃO DO PROCESSO**	consiste em desenvolver atividade para continuamente melhorar os processos de *software* usados na organização com a intenção de melhorar a qualidade de *software*, aumentar a produtividade e diminuir o ciclo de tempo do desenvolvimento do produto.

As práticas dos níveis inferiores do CMM servem como base para os níveis superiores. Implementar os níveis fora da ordem leva a riscos, os quais fazem com que as práticas sejam abandonadas ou relaxadas exatamente no instante em que elas são mais necessárias e principalmente nos momentos de crise.

A definição de processos técnicos que são previstos nas práticas do nível 3 tem poucas chances de institucionalização se as bases gerenciais do nível 2 não estiverem estabelecidas e institucionalizadas.

Podemos perceber que, conforme vamos chegando ao nível 5, o número de áreas-chave vai diminuindo. Então, é razoável concluir que as organizações que

amadureceram, principalmente nos níveis 2 e 3, tendem a realizar mais facilmente as atividades para conseguirem os níveis 4 e 5 do CMM, pois a cultura de melhoria já foi implantada e já está institucionalizada.

CONEXÃO

Recentemente apareceu o CMMI (***Capability Maturity Model Integration*** – Integração do Modelo de Maturidade e Capacidade). Trata-se de uma evolução do CMM e tem como objetivo integrar diferentes modelos e disciplinas na área de qualidade de *software*. Possui 3 modelos: CMMI-DEV (para desenvolvedores), CMMI-ACQ (para os processos de aquisição e terceirização de bens e serviços), CMMI-SVC (para empresas prestadoras de serviços).

Saiba mais sobre o CMMI, os *links* a seguir são bastante interessantes:

- http://www.blogcmmi.com.br/avaliacao/lista-de-empresas-cmmi-no-brasil: Lista das empresas no Brasil que passaram pela certificação CMMI
- http://www.cmmi.de/#el=CMMI/0/HEAD/folder/folder.CMMI: Em inglês. Trata-se de um navegador para os tópicos dos modelos do CMMI.
- http://www.blogcmmi.com.br/. É um blog com vários outros links e assuntos relacionados com CMMI.

3.3 ISO/IEC 15504 ou SPICE

A ISO/IEC 15504, também conhecida como SPICE, é a norma ISO/IEC que define processo de desenvolvimento de *software*. Ela é uma evolução da ISO/IEC 12207, mas possui níveis de capacidade para cada processo, assim como o CMMI. Em outubro de 2003, a norma ISO/IEC 15504 (SPICE) para a avaliação de processos de *software* foi oficialmente publicada pela ISO.

Esta norma define um modelo bidimensional que tem por objetivo a realização de avaliações de processos de *software* com o foco da melhoria dos processos (gerando um perfil dos processos, identificando os pontos fracos e fortes, que serão utilizados para a elaboração de um plano de melhorias) e a determinação da capacidade dos processos viabilizando a avaliação de um fornecedor em potencial. Esta norma está sendo desenvolvida desde 1993 pela ISO em conjunto com a comunidade internacional através do projeto SPICE

(*Software Process Improvement and Capability Determination*) com base nos modelos já existentes como ISO 9000 e CMM.

Segundo a norma, uma avaliação de processo de *software* é uma investigação e análise disciplinada de processos selecionados de uma unidade organizacional em relação a um modelo de avaliação de processo. A ISO/IEC 15504 define um modelo de referência de processo que identifica e descreve um conjunto de processos considerados universais e fundamentais para a boa prática da engenharia de *software*, e define seis níveis de capacidade, sequenciais e cumulativos, que podem ser utilizados como uma métrica para avaliar como uma organização está realizando um determinado processo e também podem ser utilizados como um guia para a melhoria. Veja na tabela 3.4 os seis níveis:

NÍVEL	DESCRIÇÃO
INCOMPLETO – 0	Processo não existe ou falha em atingir seus objetivos.
EXECUTANDO – 1	Processo geralmente atinge os objetivos, porém sem padrão de qualidade e sem controle de prazos e custos.
GERENCIADO – 2	Processo planejado e acompanhado que satisfaz requisitos definidos de qualidade, prazo e custos.
ESTABELECIDO – 3	Processo executado e gerenciado com uma adaptação de um processo-padrão definido, eficaz e eficiente.
PREVISÍVEL – 4	Processo executado dentro de limites de controle definidos e com medições detalhadas e analisadas.
OTIMIZADO – 5	Processo melhorado continuamente de forma disciplinada.

Tabela 3.4 – Os níveis do SPICE.

Define também um guia para a orientação da melhoria de processo, tendo como referência um modelo de processo e como uma das etapas a realização de uma avaliação de processo. Este guia sugere oito etapas sequenciais, que inicia com a identificação de estímulos para a melhoria e o exame das necessidades da organização. Em seguida, existem ciclos de melhoria, nos quais um conjunto de melhoria são identificadas, uma avaliação das práticas correntes em relação à melhoria é realizada, um planejamento da melhoria é feito, seguido pela implementação, confirmação, manutenção e acompanhamento da melhoria.

No Brasil, um modelo de *software* que tem sido bastante adotado é o MPS.BR (Melhoria de Processo de *Software* Brasileiro). Este modelo foi desenvolvido no Brasil para poder adaptar os modelos conhecidos internacionalmente para a realidade brasileira, onde a maioria das empresas de *software* são constituídas por pequenas e médias empresas. A Softex constatou em pesquisas que estas empresas, na sua maioria, apenas adotam boas práticas da engenharia de *software* quando estas são exigidas em avaliações de processos. O objetivo da Softex é fazer com que as empresas adotem sempre as boas práticas por meio do MPS.BR.

CONEXÃO

Conheça um pouco mais sobre o MPS.BR:

- http://www.softex.br/portal/softexweb/uploadDocuments/sbes2011_melhoria_processo.pdf. Excelente artigo que mostra um panorama sobre o processo de melhoria de *software* no Brasil.
- http://www.softex.br/portal/softexweb/uploadDocuments/COMPUTACAO_Brasil_Edicao_OUT-DEZ-2010.pdf. Outro bom artigo sobre o processo de melhoria de *software* no Brasil usando o MPS.BR.
- http://www.softex.br/portal/softexweb/uploadDocuments/CIBSE2010_MPSBR_CameraReady.pdf. Artigo que mostra a versão atual do MPS.BR e resultados de um estudo de caso.

3.4 ISO 12207 – Processos do Ciclo de Vida do *Software*

Este padrão formaliza a arquitetura do ciclo de vida do *software*, que é um assunto básico em engenharia de *software* e também em qualquer estudo sobre qualidade do processo de *software*.

Esta norma possui mais de 60 páginas e detalha os diversos processos envolvidos no ciclo de vida do *software*. Estes processos estão divididos em três classes: Processos fundamentais, Processos de apoio e Processos organizacionais.

Em seguida, é apresentada a lista completa dos processos.

1. Processos fundamentais: início e execução do desenvolvimento, operação ou manutenção do *software* durante o seu ciclo de vida.

a) **Aquisição**: atividades de quem um software. Inclui: definição da necessidade de adquirir um software (produto ou serviço), pedido de proposta, seleção de fornecedor, gerência da aquisição e aceitação do software.

b) **Fornecimento**: atividades do fornecedor de software. Inclui preparar uma proposta, assinatura de contrato, determinação dos recursos necessários, planos de projeto e entrega do software.

c) **Desenvolvimento**: atividades do desenvolvedor de software. Inclui: análise de requisitos, projeto, codificação, integração, testes, instalação e aceitação do software.

d) **Operação**: atividades do operador do software. Inclui: operação do software e suporte operacional aos usuários.

e) **Manutenção**: atividades de quem faz a manutenção do software.

2. Processos de apoio: Auxiliam outro processo.

a) **Documentação**: Registro de informações produzidas por um processo ou atividade. Inclui planejamento, projeto, desenvolvimento, produção, edição, distribuição e manutenção dos documentos necessários a gerentes, engenheiros e usuários do software.

b) **Gerência de configuração**: identificação e controle dos itens do software. Inclui: controle de armazenamento, liberações, manipulação, distribuição e modificação de cada um dos itens que compõem o software.

c) **Garantia da qualidade**: garante que os processos e produtos de software estejam em conformidade com os requisitos e os planos estabelecidos.

d) **Verificação**: determina se os produtos de software de uma atividade atendem completamente aos requisitos ou condições impostas a eles.

e) **Validação**: determina se os requisitos e o produto final (sistema ou software) atendem ao uso específico proposto.

f) **Revisão conjunta**: define as atividades para avaliar a situação e produtos de uma atividade de um projeto, se apropriado.

g) **Auditoria**: determina adequação aos requisitos, planos e contrato, quando apropriado.

h) **Resolução de problemas**: análise e resolução dos problemas de qualquer natureza ou fonte, descobertos durante a execução do desenvolvimento, operação, manutenção ou outros processos.

3.Processos organizacionais: implementam uma estrutura constituída de processos de ciclo de vida e pessoal associados, melhorando continuamente a estrutura e os processos.

a) **Gerência**: gerenciamento de processos.

b) **Infraestrutura**: fornecimento de recursos para outros processos. Inclui: *hardware*, *software*, ferramentas, técnicas, padrões de desenvolvimento, operação ou manutenção.

c) **Melhoria**: atividades para estabelecer, avaliar, medir, controlar e melhorar um processo de ciclo de vida de *software*.

d) **Treinamento**: atividades para prover e manter pessoal treinado.

A norma detalha cada um dos processos mostrados. Ela define ainda como eles podem ser usados de diferentes maneiras por diferentes organizações (ou parte destas), representando diversos pontos de vista para esta utilização. Cada uma destas visões representa a forma como uma organização emprega estes processos, agrupando-os de acordo com suas necessidades e objetivos.

As visões têm o objetivo de organizar melhor a estrutura de uma empresa, para definir suas gerências e atividades alocadas às suas equipes. Existem cinco visões diferentes: contrato, gerenciamento, operação, engenharia e apoio.

A ISO/IEC 12207 é a primeira norma internacional que descreve em detalhes os processos, atividades e tarefas que envolvem o fornecimento, desenvolvimento, operação e manutenção de produtos de *software*. A principal finalidade desta norma é servir de referência para os demais padrões que venham a surgir. Lançada em agosto de 1995, ela é citada em quase todos os trabalhos relacionados à engenharia de *software* desde então, inclusive aqueles relativos à qualidade.

3.5 MPS.BR

A SOFTEX (Associação para Promoção da Excelência do *Software* Brasileiro), com o apoio do Ministério da Ciência e Tecnologia (MCT), da FINEP (Financiadora de Estudos e Projetos) e do BID (Banco Interamericano de Desenvolvimento), coordena um programa para a Melhoria de Processo do *Software* Brasileiro, desde 2003, chamado de MPS.BR (Guia Geral MPS.BR, 2006, p. 4), sendo um modelo de qualidade de processos das empresas de desenvolvimento de

software, com o objetivo de melhorar os processos e serviços a médio e longo prazo, de acordo com as necessidades de negócio, buscando ainda o reconhecimento nacional e internacional como um modelo que seja aplicável à indústria de *software* e serviços (De BONA e SANTOS, 2012).

De acordo com o Guia Geral MPS.BR (2006, p. 5), para que as empresas brasileiras se posicionem em um cenário competitivo em relação à outras empresas de desenvolvimento de *software* do mundo, elas necessitam que seus empreendedores coloquem a eficiência e a eficácia dos seus processos em foco nas empresas, oferecendo produtos de *softwares* com a qualidade proporcional e conforme padrões internacionais de qualidade

Guias do MPS.BR

Busca-se que ele seja adequado ao perfil de empresas com diferentes tamanhos e características, públicas e privadas, embora com especial atenção às micros, pequenas e médias empresas. Também espera-se que o MPS.BR seja compatível com os padrões de qualidade aceitos internacionalmente e que tenha como pressuposto o aproveitamento de toda a competência existente nos padrões e modelos de melhoria de processo já disponíveis. Dessa forma, ele tem como base os requisitos de processos definidos nos modelos de melhoria de processo e atende a necessidade de implantar os princípios de Engenharia de Software de forma adequada ao contexto das empresas brasileiras, estando em consonância com as principais abordagens internacionais para definição, avaliação e melhoria de processos de software .

O MPS.BR baseia-se nos conceitos de maturidade e capacidade de processo para a avaliação e melhoria da qualidade e produtividade de produtos de *software* e serviços correlatos. Dentro desse contexto, o MPS.BR possui três componentes: Modelo de Referência (MR-MPS), Método de Avaliação (MA-MPS) e Modelo de Negócio (MN-MPS).

O MPS.BR está descrito através de documentos em formato de guias:

- **Guia Geral**: contém a descrição geral do MPS.BR e detalha o Modelo de Referência (MR-MPS), seus componentes e as definições comuns necessárias para seu entendimento e aplicação.
- **Guia de Aquisição**: descreve um processo de aquisição de *software* e serviços correlatos. É descrito como forma de apoiar as instituições que queiram adquirir produtos de *software* e serviços correlatos apoiando-se no MR-MPS.
- **Guia de Avaliação**: descreve o processo e o método de avaliação MA-MPS, os requisitos para avaliadores líderes, avaliadores adjuntos e Instituições Avaliadoras (IA).

Guia Geral MPS.BR (2006, p. 6)

O MPS.BR possui uma estrutura dividida em três partes, sendo o Modelo de Referência (MR-MPS), o Método de Avaliação (MA-MPS) e o Modelo de Negócio (MN-MPS), representados na figura 3.2. De acordo com De Bona e Santos (2012), estes modelos são descritos da seguinte forma:

- **Modelo de referência para melhoria do processo de (MR-MPS)** - responsável por mostrar às empresas os requisitos a serem cumpridos pelas organizações que têm interesse em estar em conformidade com o MR-MPS.
- **Modelo de avaliação para melhoria do processo de (MA-MPS)** - contém os processos para os avaliadores das empresas. Tem como objetivo orientar a realização de avaliações, de acordo com a norma ISO/IEC 15504.
- **Modelo de negócio para melhoria do processo de (MN-MPS)** - descreve as regras para implementação do MR-MPS pelas empresas.
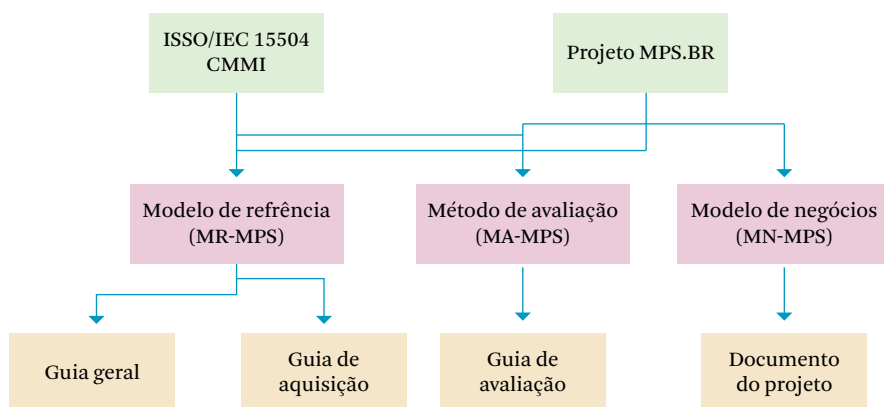
Figura 3.2 – Estrutura do MPS.BR.

O modelo MPS.BR é aderente às normas internacionais, em busca de seu reconhecimento tanto nacional quanto internacional. Além de ser compatível com o modelo CMMI-DEV, de acordo com Costa (2012, p. 12) "o modelo de referência MR-MPS contém as definições dos processos com base na norma ISO/IEC 12207:2008 e estabelece níveis de maturidade de processos com base nos perfis de capacidade estabelecidos na norma ISO/IEC 15504".

O MR-MPS (Modelo de Referência para Melhoria do Processo de *Software*) apresenta sete níveis de maturidade, descritos, a seguir, a partir de o mais alto nível para o mais baixo:

- A (Em otimização);
- B (Gerenciado quantitativamente);
- C (Definido);
- D (Largamente definido);
- E (Parcialmente definido);
- F (Gerenciado);
- G (Gerenciado parcialmente).

Em cada nível maturidade relacionado é atribuído um perfil de processos e de capacidade de processos, conforme a norma ISO/IEC 15504, permitindo que a organização realize suas implementações dos processos de modo mais gradual, proporcionando o amadurecimento concomitante a estas implementações (COSTA, 2012, p. 12).

Diferentemente do CMMI, que possui apenas cinco níveis, os sete níveis do MPS.BR possibilitam que as empresas realizem sua implementação de maneira mais gradual e adequada, principalmente a pequenas empresas, simplificando o controle e a medição do quanto esta empresa está dominando os processos de *software*, já que a maturidade nestas empresas está mais associada ao conhecimento que ela possui e como esses conhecimentos são aplicados (De BONA e SANTOS, 2012). O comparativo entre estes dois modelos pode ser visto na figura 3.3, onde observa-se o relacionamento entre o MPS.BR e o CMMI.
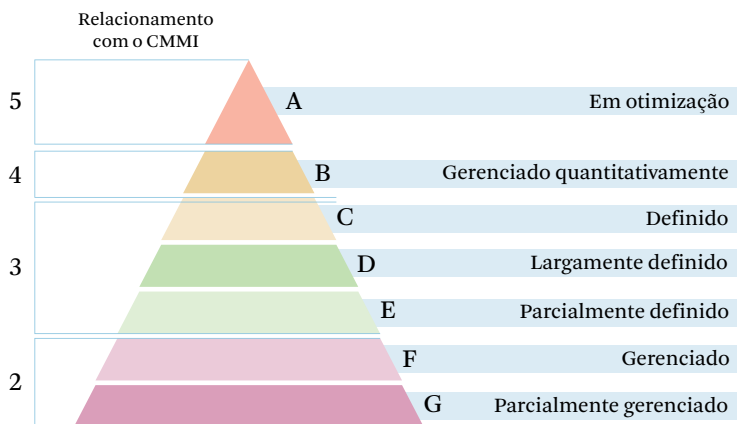
Figura 3.3 – Níveis do MR-MPS e relacionamento com o CMMI. Disponível em:<http://www.fumsoft.org.br/wp-content/uploads/2014/07/tabela_reduzida.jpg>.

Cada nível possui um perfil de processos e capacitação de processos definidos, conforme demonstra De Bona e Santos (2012) na tabela a seguir:

NÍVEL G	Há uma gerência de projetos e uma gerência de requisitos, o que demonstra um nível de maturidade bastante baixo. Porém, como se sabe, na gerência de projetos existem algumas tarefas essenciais, que precisam ser realizadas em qualquer projeto de *software*, como identificação e monitoramento de atividades referentes ao projeto. No que diz respeito à gerência de requisitos, quer dizer que a empresa tem uma boa habilidade de lidar com as mudanças que os requisitos sofrem ao longo do desenvolvimento de um projeto.
NÍVEL F	Há uma maturidade um pouco maior por parte da empresa, existindo uma garantia da qualidade do produto, bem como controle nas aquisições (de produtos e serviços), gerência de configuração e medição. A primeira diz respeito a garantir que os produtos e processos estejam de acordo com o plano e recursos predefinidos. Isso não garante que o *software* será um sucesso, porém é o primeiro passo para tal. Já o controle nas aquisições diz respeito à definição de necessidades de aquisição, bem como definição de fornecedores e outros pontos importantes para a compra de produtos e/ou serviços. A gerência de configuração é responsável por estabelecer e controlar os resultantes de um processo, enquanto a medição diz respeito à coleta e análise dos dados relativos aos produtos desenvolvidos, bem como aos processos implementados.
NÍVEL E	Garante uma organização ainda maior nos processos. Ele trás a adaptação do processo para gerência de projeto, a definição do processo organizacional, bem como sua avaliação e melhoria, trazendo também o processo de treinamento dentro da empresa de forma global. O primeiro processo diz respeito a estabelecer e gerenciar o projeto, com envolvimento dos clientes e de acordo com princípios predefinidos. Já o segundo e terceiro dizem respeito à organização da empresa no que diz respeito a fazer *software*. Eles são responsáveis por definir esses processos, bem como avaliá-los e melhorá-los.
NÍVEL D	É o nível de maturidade intermediário, trazendo algumas ideias de desenvolvimento de requisitos, bem como utilização de técnicas para garantir a qualidade do *software*, como validação e verificação. Nesse nível de maturidade, a empresa é capaz de coletar os requisitos juntamente com o cliente, deixando tudo largamente definido a priori. Posteriormente, há a necessidade de confirmar se esses requisitos foram atingidos, o que trás a solução técnica (onde há a especificação técnica dos requisitos validados), a validação, verificação, a integração e instalação do produto, bem como sua liberação.
NÍVEL C	Traz alguns processos mais avançados, como gerência de riscos e análise de decisão e resolução. O primeiro diz respeito a diminuir os riscos que mudanças de pessoal possam trazer para os projetos, evitando atrasos no cronograma. Esse é um ponto muito importante, a que todas as empresas devem estar atentas, uma vez que o cliente quer que o cronograma seja cumprido. Já a análise de decisão e resolução é responsável por analisar as decisões que foram tomadas utilizando um processo formal de avaliação.
NÍVEL B	Traz uma gerência quantitativa do projeto, além de calcular o desempenho do processo organizacional. O propósito da gerência quantitativa é bem interessante, e diz respeito a monitorar e determinar os projetos em relação aos seus objetivos de qualidade e desempenho, garantindo que o *software* tenha qualidade. O desempenho do processo organizacional é calculado para que se tenha um entendimento da qualidade dos processos que a organização está realizando.
NÍVEL A	Há uma maturidade plena da empresa. Ele trás processos de análise de causas e resolução e inovação na organização. O objetivo do primeiro é identificar e, obviamente, evitar no futuro causas de defeitos, seja nos processos ou nos projetos realizados. Já a inovação é parte essencial em qualquer empresa consolidada no mercado, trazendo a ideia de sempre buscar melhorias e evolução para melhor.

Figura 3.5 – Os níveis de maturidade do MR-MPS. De BONA e SANTOS (2012).

O nível de maturidade é muito importante para que a empresa compreenda, juntamente com seus clientes, onde ela está posicionada e o que pode ser melhorado em relação aos seus processos, atentando para a evolução constante em meio às adaptações necessárias. O nível A é um exemplo típico de empresas que possuem seus processos consolidados, mas que mesmo assim buscam melhorar ainda em busca da excelência permanente, buscando sempre inovações e novidades para oferecer ao mercado. Por outro lado, as empresas no nível G, em sua maioria, estão começando esta caminhada de definição dos processos e melhorias em suas práticas, ou ainda não tiveram êxito ou tempo suficiente para evoluí-los.

De acordo com Wikipédia (2015), cada nível de maturidade do MPS.BR, no MR-MPS, possui suas áreas de processo, onde são analisados:

- processos fundamentais
    - aquisição;
    - gerência de requisitos;
    - desenvolvimento de requisitos;
    - solução técnica;
    - integração do produto;
    - instalação do produto;
    - liberação do produto.

- processos organizacionais
    - gerência de projeto;
    - adaptação do processo para gerência de projeto;
    - análise de decisão e resolução;
    - gerência de riscos;
    - avaliação e melhoria do processo organizacional;
    - definição do processo organizacional;
    - desempenho do processo organizacional;
    - gerência quantitativa do projeto;
    - análise e resolução de causas;
    - inovação e implantação na organização.

- processos de apoio
  - garantia de qualidade;
  - gerência de configuração;
  - validação;
  - medição;
  - verificação;
  - treinamento.

No MA-MPS (Método de avaliação para melhoria do processo de *software*), a avaliação está assim estruturada, de acordo com Wikipédia (2015):

- **Avaliação MA-MPS**
  - Equipe de avaliação: 3 a 8 pessoas, sendo:
    - 1 avaliador líder
    - no mínimo 1 avaliador adjunto
    - no mínimo 1 técnico da empresa
  - Duração: 2 a 4 dias;
  - Validade: 3 anos;

- **Estruturação da Avaliação:**
  - Planejar e preparar avaliação
    - Plano de Avaliação / Descrição dos indicadores de processo;
  - Conduzir Avaliação
    - Resultado da avaliação;
  - Relatar resultados
    - Relatório da avaliação;
  - Registrar e publicar resultados
    - Banco de dados Softex

O MPS.BR tem sido um verdadeiro sucesso e ainda fornece diversos meios para que se difunda de maneira estruturada, por meio de provas oficiais, cursos e capacitando profissionais. De acordo com o *site* oficial do projeto (www.softex.br/mpsbr), há quase 6 mil profissionais capacitados no mercado, e a tendência é de que esse número cresça muito nos próximos anos, conforme afirmam De Bona e Santos (2012). Portanto, vemos que o MPS.BR é um modelo que vem adotando as melhores práticas de Engenharia de *Software*, sendo uma grande

vantagem tanto do ponto de vista técnico, como do ponto de vista econômico, pelo fato de ser mais barato que os demais modelos existentes no mercado.

ATIVIDADES

01. Quais seriam os impactos de se adotar uma norma em uma empresa de desenvolvimento de *software*?

02. Quais são os níveis do CMM/CMMI? Explique cada um de seus níveis.

03. Por que é tão difícil conseguir e estabelecer o nível 5 do CMM em uma empresa? Explique.

04. Cite as 3 guias (documentos) do MPS.BR e explique cada uma delas.

05. Pesquise sobre as áreas-chave (KPA) do CMM e apresente-as de acordo com cada nível.

REFLEXÃO

Neste capítulo vimos que existem vários modelos que podem ser usados e isto pode ser um difícil trabalho para os profissionais de TI escolher entre esses modelos e padrões e selecionar o melhor a ser aplicado na situação necessária.

Porém, não é uma escolha exclusivamente do setor de TI, é uma decisão conjunta com outros setores da empresa e que deve envolver a alta direção, em um trabalho corporativo conjunto. Logo, mais uma vez fica clara a importância fundamental da TI nas empresas.

Mas, a realidade brasileira ainda é diferente dos exemplos que encontramos nas bibliografias e estudos de caso. A grande maioria das empresas de *software* nacional ainda é de pequeno ou médio porte e não possuem uma estrutura tão grande para implantar modelos de qualidade de *software* como os mostrados aqui. Neste caso, modelos como o MPS/BR são usados e largamente difundidos entre os grupos de *software* nacional e contribuem como mais uma variável para o gestor de TI ter que escolher entre os modelos existentes.

Desta forma, podemos ver que os modelos de processos para a qualidade de *software* são excelentes, porém necessitam de um detalhe corporativo e institucional para ser utilizado na sua plenitude: ter a participação de toda a empresa e um bom patrocinador interno, de preferência, proveniente da diretoria da empresa.

LEITURA

- Sobre o MPS.BR existem alguns artigos interessantes disponíveis na Internet, bem como o Guia Geral que contém a descrição geral do MPS.BR e detalha o Modelo de Referência (MR -MPS) e as definições comuns necessárias para seu entendimento e aplicação, que pode ser encontrado aqui: http://migre.me/oJyRV
- Uma abordagem para condução de iniciativas de melhoria de processos de *software*, disponível em: http://migre.me/oJyDC
- Gerenciamento da Qualidade no Desenvolvimento de Sistemas: Plano de Transição do nível 2 do SW-CMM para o nível 2 do CMMI-DEV em uma Empresa de Tecnologia: http://migre.me/oJyOw
- Documento da ISO/IEC15504, disponível em: http://migre.me/oJz5S
- O MC2Q-SQ - Um Método para a Melhoria Contínua da Qualidade de *Software* apoiado em CMMe SPICE - e sua Aplicação Experimental como base nos Programas de Melhoria da Qualidade de *Software*, disponível em: http://migre.me/oJzdM

REFERÊNCIAS BIBLIOGRÁFICAS

COSTA, T. M., **Melhoria Contínua de Processo de Software utilizando a teoria das restrições**. Dissertação de Mestrado UFRJ. Rio de Janeiro, 2012. p. 233.

De BONA, G.; SANTOS, I. R. **Gerenciando requisitos com MPS.BR**. Revista Engenharia de Software, Ed. 63, 2012. Disponível em: <http://www.devmedia.com.br/gerenciando-requisitos-com-mps-br/29375>. Acesso em: 15 dez. 2014.

Guia Geral MPS.BR, 2006. **Associação para Promoção da Excelência do Software Brasileiro** SOFTEX. MPS.BR - Melhoria de Processo de Software Brasileiro - Guia Geral. 2. ed. 2006.

SILVA, B. C. C. **Integrando Agilidade com maturidade**. Revista Engenharia de Software Magazine. Ed. 59. Disponível em: <http://www.devmedia.com.br/integrando-agilidade-com-maturidade-revista-engenharia-de-software-magazine-59/28197>. Acesso em: 16 dez. 2014.

WIKIPÉDIA. **Melhoria de Processos do Software Brasileiro**. Disponível em: <http://pt.wikipedia.org/wiki/Melhoria_de_Processos_do_Software_Brasileiro>. Acesso em: 02 fev. 2015.

4

Modelos de Ciclo de Vida de *Software*

Neste capítulo veremos sobre alguns dos principais modelos de ciclo de vida de *software* e suas principais características. Ciclos de vida de *software* representam as etapas ou fases que um *software* passa desde a sua concepção até a sua retirada. É de extrema importância compreendermos estas fases e seus processos relacionados, bem como perceber a existência de diferentes abordagens.

Como o desenvolvimento de *softwares* é uma tarefa complexa e a cada dia os clientes e o próprio negócio exigem maior qualidade no produto final entregue e com o menor tempo possível, várias metodologias tentam impulsionar ou organizar estas atividades e processos dentro da Engenharia de *Software*. Métodos clássicos da engenharia, com um processo linear, recheado de planejamento, especificação de requisitos bem definidos e uma pesada documentação têm sido sufocados pela pressão do tempo de entrega e pela concretização daquilo que estava no papel. Em virtude da evolução destes processos e diferentes metodologias, existem métodos com uma abordagem ágil. Processos ágeis ilustram outras características diferentes dos métodos tradicionais e também possuem grandes desafios para serem devidamente implementados nas organizações.

Os processos de *software* são melhorados sempre para aumentar a qualidade do produto final com redução de custo e tempo. Assim, este capítulo abordará os modelos de ciclo de vida de *software*, ou seja, os seus modelos de processos, envolvendo o modelo cascata, o processo iterativo e o processo ágil, entre os vários modelos existentes.

OBJETIVOS

- Aprender sobre modelos de ciclo de vida de *software*;
- Conhecer o processo Cascata e suas principais utilidades;
- Ter uma visão geral sobre processos Iterativos;
- Conhecer a metodologia ágil e seus processos.

4.1 Introdução

Todas as vezes que a palavra *software* é vista ou ouvida, associamos este termo a um programa de computador. Porém, o *software* não é apenas o programa, mas sim uma série de documentos, modelos de dados, configuração, requisitos etc., que o compõe. Esses elementos são chamados de artefatos de *software*.

Os engenheiros de *software*, que são os responsáveis pela manutenção destes artefatos, usam várias técnicas para gerá-los, e estas provém de outras semelhantes às empregadas na engenharia tradicional.

Existem, na verdade, dois tipos principais de produtos de *software* segundo Sommerville (2007):

- **Produtos genéricos**: esses produtos são aqueles encontrados disponíveis para venda em vários tipos de mídias. São também chamados de "*software* de prateleira". Como exemplo podemos citar os pacotes para escritório, editoração gráfica, sistemas operacionais comerciais etc.
- **Produtos sob encomenda**: esses produtos são aqueles desenvolvidos para uma determinada finalidade encomendada pelo cliente. Como exemplo, podemos citar os *softwares* para dispositivos embarcados, para um determinado processo de negócio.

Porém, nos principais sistemas ERP e outros, são possíveis ter um tipo de produto "híbrido", ou seja, um produto que foi desenvolvido por uma empresa para ser solução para várias áreas de negócio de uma empresa, mas também pode sofrer adaptações para suprir uma determinada especificidade em uma situação ou processo de negócio.

Este tipo de situação é chamado de customização de módulos.

A engenharia de *software* é, portanto, uma área da engenharia aplicada à produção de *software*.

Porém, a engenharia de *software* pode ser confundida com uma disciplina chamada engenharia de sistemas. Há uma pequena diferença entre as duas: a engenharia de sistemas trata exclusivamente da área de desenvolvimento de sistemas e seus aspectos, incluindo *hardware*, *software* e engenharia de processo. A engenharia de *software* é uma parte desse processo.

Entre os principais conceitos a serem aprendidos em engenharia de *software*, o processo de *software* é um dos principais.

Um processo de *software*, segundo Sommerville (2007), é um conjunto de atividades que leva à produção de um produto de *software*.

Essas atividades envolvem o desenvolvimento do *software* usando alguma linguagem de programação. O processo de *software* é um processo intelectual e criativo, sendo, assim, que a possibilidade de automatizá-los é um pouco limitada. O uso de ferramentas automatizadas em engenharia de *software* é bastante comum e existe uma ferramenta específica chamada CASE (Computer Aided *Software* Engineering – Engenharia de *Software* Auxiliada por Computador) que apoia algumas atividades do processo. No entanto, muitas ainda não são automatizadas. Um dos problemas da limitação das ferramentas CASE é que existem muitos tipos de processo de *software* e não é possível criar uma ferramenta que abranja todos ou a grande maioria.

Além disso, nenhum processo de *software* é ideal e existem muitas organizações que desenvolveram abordagens bem diferentes para o desenvolvimento de *software*. Desta forma, a diversidade de formas de desenvolvimento não permite modelar uma ferramenta computacional que abranja os métodos existentes.

> Ainda segundo Sommerville (2007), uma ferramenta CASE é o nome dado a um *software* usado para apoiar as atividades de processo de *software*. Entre os processos, podem ser: engenharia de requisitos, projeto, desenvolvimento de programas, teste, modelagem de dados, entre outros. A ferramenta oferece automatização de várias tarefas. Por exemplo, existem ferramentas CASE que usam gráficos e diagramas para representar os módulos de um *software* e sua interconexão, oferecendo até mesmo assistentes para a geração do código-fonte que implementará os diagramas desenvolvidos pelo engenheiro de *software*.

Existem vários processos de *software* diferentes, como já citado, porém algumas atividades são fundamentais e comuns a todos eles. Estas atividades são:

- **Especificação de *software***: essa atividade envolve a definição de funcionalidades e as restrições existentes na sua operação.
- **Projeto e implementação**: essa atividade abrange as questões relativas à produção de um *software* que atenda à especificação inicial.
- **Validação de *software***: são as atividades que garantem que o *software* tenha validade junto ao cliente

- **Evolução de *software***: são as atividades que garantem que a evolução do *software* cresça à medida que as necessidade do cliente fiquem diferentes.

Como já foi citado, existem vários processos de *software* e sua diversidade é bem vasta porém, eles precisam ser aprimorados, e para isso a padronização desses processos é necessária.

CONEXÃO

A seguir alguns *links* referentes às ferramentas case existentes no mercado. Além dessas, obviamente existem outras, porém os *links* abaixo são o início de outras pesquisas que você pode fazer.

- http://www.devmedia.com.br/post-2174-Ferramentas-case-parte-iv.html: Artigo exemplificando o Microsoft Visio como ferramenta CASE.
- http://www.casestudio.com/enu/default.aspx. Site do *software* Case Studio. Trata-se de um *software* bastante popular para o apoio às atividades de desenvolvimento. Em inglês.
- http://erwin.com/. O ErWin é outro exemplo de ferramenta CASE bastante usada no mercado. Em inglês.

Padronizando processos, a comunicação é melhorada, o tempo de treinamento é reduzido e o apoio a um processo automatizado fica mais econômico. A padronização contribui também para a introdução de novos métodos e técnicas de engenharia de *software* e, consequentemente, do uso de melhores práticas na arte da engenharia de *software*.

Durante esta disciplina, vamos ver estes tópicos com um pouco mais de detalhes.

Vamos começar conceituando o ciclo de vida do *software*.

4.1.1 Ciclo de vida do *software*

Segundo Gustafson (2003), o ciclo de vida de um *software* é uma sequência de diferentes atividades executadas durante o desenvolvimento do *software*.

A interação entre essas atividades pode ser definida segundo diferentes modelos ou processos de desenvolvimento. Os modelos existentes possuem diferentes graus de sofisticação e complexidade. Para projetos que envolvem uma

equipe de desenvolvimento pouco numerosa e experiente, o mais adequado seria, provavelmente, um processo simples. No entanto, para sistemas maiores que envolvem equipes de dezenas ou centenas de elementos e milhares de utilizadores, um processo simples não é suficiente para oferecer a estrutura de gestão e disciplina necessárias à engenharia de um bom produto de *software*.

Desta forma, é necessária algo mais formal e disciplinado. É importante fazer notar que isto não significa que se perca em inovação ou que se ponham entravés à criatividade. Significa apenas que é utilizado um processo bem estruturado para permitir a criação de uma base estável para a criatividade.

Por mais simples ou complexo que possa parecer, um modelo de ciclo de vida de um projeto é, de fato, uma versão simplificada da realidade. É suposto ser uma abstração e, tal como todas as boas abstrações, apenas a quantidade de detalhe necessária ao trabalho em mãos deve ser incluída. Qualquer organização que deseje pôr um modelo de ciclo de vida em prática precisará adicionar detalhes específicos para dadas circunstâncias e diferentes culturas.

Por exemplo, a Microsoft quis manter uma cultura de pequena equipe e ao mesmo tempo tornar possível o desenvolvimento de grandes e complexos produtos de *software*.

A engenharia de *software* tem produzido inúmeros modelos de ciclo de vida, entre eles os modelos de cascata, espiral e desenvolvimento rápido de aplicações (RAD). Antes do modelo de cascata ser proposto em 1970, não havia concordância em termos dos métodos a levar a cabo no desenvolvimento de *software*. Desde então, ao longo dos anos, muitos modelos foram e são propostos refletindo, assim, a grande variedade de interpretações e caminhos que podem ser tomados no desenvolvimento de *software*.

4.1.2 Crise do *software*

A engenharia de *software* é uma área da computação que praticamente nasceu junto com a própria computação.

Devido ao crescente uso da computação, a demanda por *software* cresceu conjuntamente, porém a forma na qual o *software* era desenvolvido na década de 1970 era desordenada e sem critérios ou técnicas adequadas.

Frente a isso, a demanda por soluções computacionais aumentava, a complexidade dos problemas também e as técnicas de desenvolvimento não eram

suficientes para tal. Logo, é de se esperar que ocorra uma "crise" entre os componentes deste ambiente. Esta crise foi chamada de Crise do *Software* e foi nomeada assim pela primeira vez por Edsger Dijkstra em um artigo de 1972. (DIJKSTRA, 1972).

Existem várias causas para a crise do *software* e cada autor a trata de uma maneira diferente, porém basicamente, temos as seguintes principais causas relacionadas com a crise do *software*:

- os problemas a serem resolvidos estavam se tornando cada vez mais complexos;
- não existiam técnicas adequadas para o desenvolvimento e solução dos problemas;
- não existiam técnicas que pudessem validar o *software* que foi desenvolvido;
- o processo de desenvolvimento do *software* também estava se tornando complexo.

A crise do *software* pode ser percebida de diversas formas, principalmente em situações nas quais as falhas do *software* aparecem, por exemplo quando:

- o orçamento não é suficiente e acaba estourando;
- o prazo é curto demais e acaba estourando;
- o *software* não atende aos requisitos, demonstrando ter baixa qualidade (vamos ver isso com mais detalhes à frente);
- o processo de desenvolvimento de *software* é difícil de ser gerenciado;
- a manutenção do *software* fica difícil de ser efetuada, principalmente nas questões relacionadas com a codificação;
- há problemas na documentação do *software*: é ruim, é falha, é pouca, é imprecisa, é inadequada etc.

O interessante é que vários projetos de *software* continuam com os mesmos problemas nos dias atuais. Quer dizer, um problema detectado na década de 1960, quando a tecnologia que apoiava era praticamente inexistente, é perpetuado até os dias atuais, com toda a tecnologia à disposição.

A Standish Group, uma empresa que trabalha principalmente com projetos de *software*, publicou em 2009 um artigo chamado Chaos Report

(Relatório do Caos), segundo o *site* Project Smart (Dominguez, 2009). Esse relatório é uma publicação frequente e realizada bianualmente. No artigo que faz referência ao relatório de 2009 foi ressaltado que os projetos considerados com sucesso caíram de 36% para 32% entre os dados de 2009 e 2002. Observe a tabela 4.1 obtida do artigo:

	1994	1996	1998	2000	2002	2004	2006	2009
SUCCESFUL	16%	27%	26%	28%	34%	29%	35%	32%
CHALLENGED	53%	33%	46%	49%	51%	53%	46%	44%
FAILED	31%	40%	28%	23%	15%	18%	19%	24%

Tabela 4.1 – Evolução da taxa de sucesso, falhas e mudanças. Fonte: DOMINGUEZ (2009).
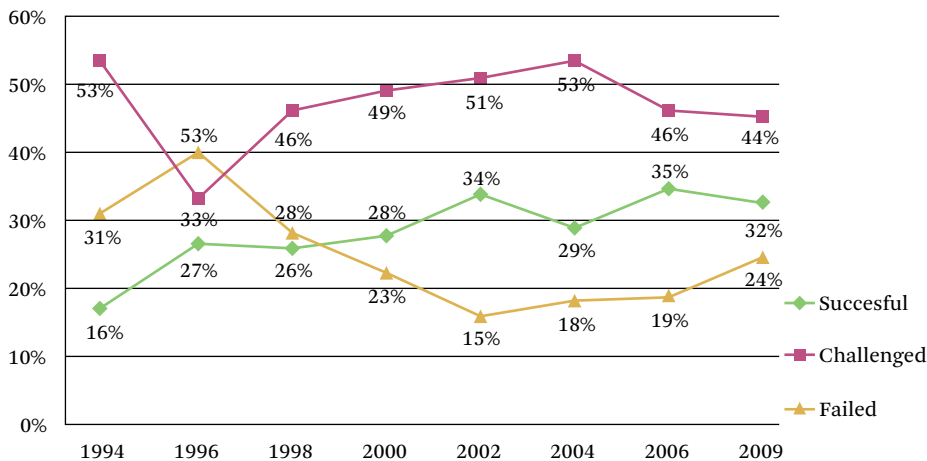
Figura 4.1 – Evolução gráfica dos índices da tabela 4.1.

Como podemos perceber pela figura 4.1, os dados segundo as pesquisas deste *site* refletem um aumento tímido nos índices de sucesso dos projetos em TI e uma queda drástica nos índices de falhas; enfim, é uma evolução equilibrada entre os índices. Logo, é mais um fato que colabora para a crise do *software*.

Segundo o Standish Group, as definições para os resultados são:

• **Tipo 1 ou Sucesso** – o projeto é completado dentro do prazo e orçamento e entrega todos os requisitos inicialmente propostos.

• **Tipo 2 ou Challenged (Mudança)** – o projeto é finalizado e fica operacional, mas estoura o orçamento e o prazo e oferece algumas das características inicialmente propostas.

• **Tipo 3 ou Fracasso** – o projeto literalmente fracassa ou é até mesmo cancelado.

Os relatórios apresentados pelo Standish Group são muito úteis para uma análise histórica desses fatos. De qualquer forma, é um valor quantitativo e isso é importantes para considerar qualquer tipo de análise. Porém, há quem os critique devido aos critérios que são utilizados para a elaboração desses relatórios. O artigo The Rise and Fall of the Chaos Report Figures (EVELEENS & VERHOEF, 2010) é um exemplo desses críticos. Segundo os autores, o Standish Group corrompe os dados e os torna unilaterais, confundem as práticas usadas nas estatísticas e, consequentemente, os resultados se tornam errados. Os críticos alegam que os dados que o Standish Group apresentam não são validados, pois não são divulgadas suas origens. Então, a única forma de validar os dados é obter outra amostra e realizar a pesquisa novamente. Porém, mesmo com estas questões, os dados do Chaos Report são úteis como informações de referência e histórico.

O departamento de defesa e o governo americano começaram a perceber que os problemas existentes na época com o desenvolvimento de *software* certamente afetaria o futuro. Ainda mais que, com o crescente uso do *software* nas organizações, certamente a economia do país seria muito afetada com as consequências ruins em relação ao modo de como os processos de *software* estavam sendo feitos.

Devido a esses problemas, o governo americano criou em 1984 o SEI (*Software* Engineering Institute), vinculado à Universidade Carnegie Mellon. Esse instituto tem o objetivo de realizar estudos na área de engenharia de *software* e, desde então, é considerado uma referência na área.

A conclusão do governo americano junto com o SEI é que para ter um futuro melhor em relação ao *software* seria necessário organizar o presente. De acordo com Dijkstra (DIJKSTRA, 1972), a causa da crise do *software* e sua relação com o futuro estão relacionados com o seguinte parágrafo do seu artigo *The humble programmer*:

> “A maior causa da crise do software é que as máquinas tornaram-se várias ordens de magnitude mais potentes! Em termos diretos, enquanto não havia máquinas, programar não era um problema; quando tivemos computadores fracos, isso se tornou um problema pequeno, e agora que temos computadores gigantescos, programar tornou-se um problema gigantesco.”

Portanto, com base no que foi visto até agora, e contextualizando os problemas da época que vimos, uma forma de controlar a crise do *software*, de acordo com Bauer, seria o estabelecimento de usos da engenharia tradicional no desenvolvimento de *software*. Surgiu a primeira definição de Engenharia de *Software* e esta é utilizada até hoje: “O estabelecimento e o uso de sólidos são os princípios de engenharia para que se possa obter economicamente um *software* que seja confiável e que funcione eficientemente em máquinas reais” (BAUER, 1977).

4.2 Processo Cascata *(Water Fall)* ou TOP DOWN

4.2.1 Ciclo cascata ou modelo clássico

Segundo (Sommerville, 2007), o primeiro ciclo de vida publicado é o modelo clássico ou também chamado de ciclo cascata (por ser uma série de eventos sequenciais ordenados). Este modelo foi criado por Winston W. Royce em 1970 e aperfeiçoado por Barry Boehm em 1976.

Até a década de 1980, foi o único modelo mais aceito pela comunidade. O modelo, de acordo com suas características, é derivado das práticas de engenharia tradicional, desta forma existe um estabelecimento de uma ordem no desenvolvimento de grandes produtos de *software*. Trata-se de um modelo mais rígido e menos administrativo, quando comparado com outros.
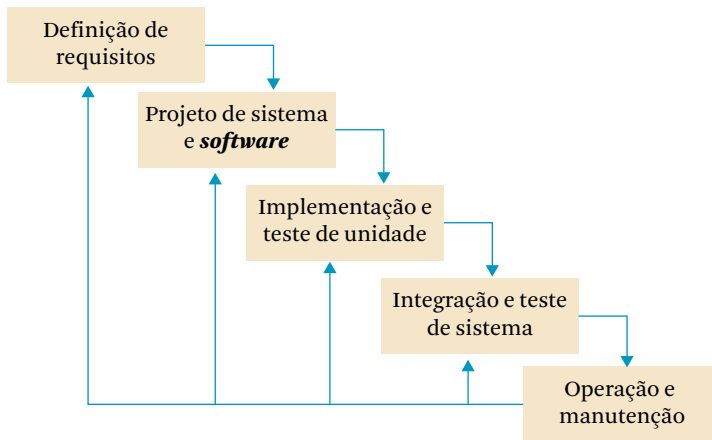
Figura 4.2 – Modelo em Cascata. Fonte: SOMMERVILLE (2007).

Conforme mostrado na figura 4.2, o modelo consiste de cinco etapas principais.

• **Análise e definição de requisitos**: os usuários do sistema são consultados de forma a elucidar quais são os serviços, restrições e objetivos do sistema. Assim que são definidos, servem como uma especificação do sistema.

• **Projeto de sistema e *software***: nesta etapa do processo, o sistema é dividido em requisitos de *hardware* e/ou de *software*. Também é estabelecida uma arquitetura geral do sistema. Nesta etapa, as abstrações fundamentais do sistema são estabelecidas, identificadas e descritas.

• **Implementação e teste de unidade**: esta etapa compreende a parte de programação do sistema e os testes das unidades desenvolvidas. As partes do *software* nesta etapa são chamadas de unidades de programa. O teste unitário serve para testar cada unidade e validá-la separadamente.

• **Integração e teste do sistema**: uma vez que as unidades de programa foram desenvolvidas e testadas, nesta etapa elas são integradas como um todo e os testes de sistema são realizados. Desta forma, os requisitos inicialmente estabelecidos são verificados e validados. Uma vez cumpridos, o *software* é liberado para o cliente.

• **Operação e manutenção**: normalmente esta é a fase mais longa do ciclo de vida. O sistema é colocado em operação e, desta forma, os erros que aparecerem compreenderão o trabalho de manutenção.

O modelo em cascata é importante porque ele serve de inspiração e referência para outros modelos, inclusive os mais modernos. Atualmente ainda é bastante lógico utilizá-lo e muitas equipes ainda o adotam.

Este modelo é orientado para a documentação detalhada de cada etapa. A documentação não compreende apenas componentes textuais, mas sim representações gráficas de modelos de processo, banco de dados, responsabilidades e até mesmo algumas simulações.

Os nomes das etapas do modelo em cascata apresentados na figura 4.2 podem mudar de autor para autor, porém como vimos, as etapas são lineares: uma é realizada após a finalização da antecedente, e isto pode ser considerada uma desvantagem do modelo. Observando a figura 4.2, se os requisitos não estiverem muito bem elaborados e claros, as demais fases estarão seriamente comprometidas.

Além disso, os requisitos, dependendo do projeto, estão em constante mudança, Logo, a etapa de requisitos não pode ficar muito tempo parada enquanto o desenho e implementação do sistema são completados. Então, como mostra a figura 4.2, as setas que voltam para as etapas iniciais servem como um *feedback* e forma de melhoria.

Porém, a constante interação descaracteriza o modelo em cascata. A ideia inicial do modelo é ser linear, mas como foi dito, existem as variações do modelo que permitem interações. As revisões e avaliações proporcionadas pelo *feedback* comentado acima referem-se ao processo e não necessariamente proveniente de usuários.

Mesmo sendo um modelo bastante popular, pode-se apontar algumas limitações apresentadas por este modelo:

- o modelo assume que os requisitos são inalterados ao longo do desenvolvimento; isto, em boa parte dos casos, não é verdadeiro, uma vez que nem todos os requisitos são completamente definidos na etapa de análise;
- muitas vezes, a definição dos requisitos pode conduzir à definição do *hardware* sobre o qual o sistema vai funcionar; dado que muitos projetos podem levar diversos anos para serem concluídos; estabelecer os requisitos em termos de *hardware* é um tanto temeroso, dadas as frequentes evoluções no *hardware*;
- o modelo impõe que todos os requisitos sejam completamente especificados antes do prosseguimento das etapas seguintes; em alguns projetos, é

às vezes mais interessante poder especificar completamente somente parte do sistema, prosseguir com o desenvolvimento do sistema, e só então encaminhar os requisitos de outras partes; isto não é previsto a nível do modelo;

• as primeiras versões operacionais do *software* são obtidas nas etapas mais tardias do processo, o que, na maioria das vezes, inquieta o cliente, uma vez que ele quer ter acesso rápido ao seu produto (MAZZOLA, 2010, p.13).

Marcoratti (2014) comenta que um dos riscos desta abordagem em caso de ausência de um processo de gestão do projeto e de controle das alterações bem definido, "podemos passar o tempo num ciclo infinito, sem nunca atingir o objetivo final, ou seja, disponibilizar o sistema a funcionar", e ainda aponta as principais desvantagens deste modelo:

• Dificuldade em acomodar mudanças depois que o processo está a ser executado;
• Partição inflexível do projeto em estágios distintos;
• Dificuldade em responder a mudanças dos requisitos;
• É mais apropriado quando os requisitos são bem compreendidos;
• Os projetos reais raramente se adaptam ao modelo linear e sequencial;
• É difícil capturar os requisitos de uma só vez;
• Cliente tem de pacientemente esperar o resultado final;
• Os programadores são frequentemente atrasados sem necessidade;
• Alto custo de correção das especificações quando nas fases de Teste e Implantação.

4.3 Processo iterativo

Este modelo combina as vantagens do modelo cascata e prototipação, devido a limitações que o modelo cascata apresenta, trazendo a ideia central do desenvolvimento incremental, sendo que a cada incremento são adicionadas novas funções ao sistema até atingir os objetivos finais, efetuando-se adaptações e mudanças necessárias a cada passo realizado (MAZZOLA, 2010, p.14). Este modelo iterativo pode ser representado por meio da figura 3 abaixo.
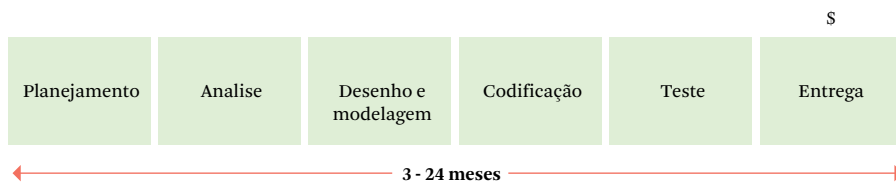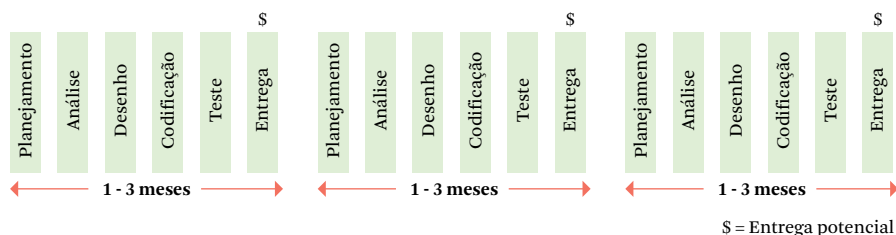
Figura 4.3 – Esquema comparativo entre o modelo cascata (a) e o processo iterativo (b). Fonte: Shore e Warden (2008, p. 35) (Adaptado pelo autor).

O desenvolvimento incremental foi proposto por H. Mills em 1980 como forma de redução do retrabalho no processo de desenvolvimento e por permitir que os requisitos pudessem ser definidos em outro momento, sem a necessidade de ter a definição completa logo no início do projeto (SOMMERVILLE, 2006, p.43).

Nesta abordagem, o desenvolvimento incremental possui algumas características marcantes, como, por exemplo, o desenvolvimento do sistema por partes; classificação dos requisitos por ordem de prioridade, identificados pelos clientes; definição do número e tamanho dos incrementos; utilização do processo de desenvolvimento preferido para estes incrementos e adição de um novo requisito à cada novo incremento, conforme pode ser visto no esquema da figura 4.4.
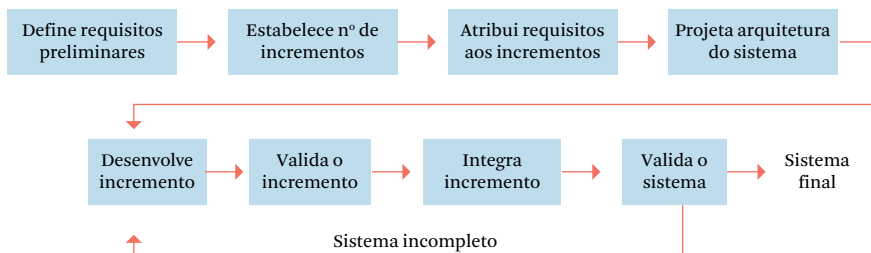
Figura 4.4 – Desenvolvimento incremental. Fonte: SOMMERVILLE (2006, p.43)

Uma vez que os incrementos a serem implementados são identificados, são detalhados os requisitos para as funções a serem entregues, e esta implementação é efetuada pelo processo de desenvolvimento que for mais apropriado. Durante o desenvolvimento destes incrementos, podem surgir outras análises de requisitos, mas as mudanças ou alterações só ocorrem no próximo incremento e não no atual (SOMMERVILLE, 2006, p.44).

Depois de concluído e entregue o incremento, este pode ser colocado em operação pelos clientes, o que vale notar que estes clientes podem receber com antecedência uma parte da funcionalidade do sistema. Com isso, estes podem testar e experimentar o sistema, oferecendo condições de levantar outros requisitos para incrementos seguintes, proporcionando um processo de melhoria a cada entrega. As funcionalidades podem ser implementadas no início do processo ou de forma gradativa, à medida que surge a necessidade ou que estas sejam exigidas (SOMMERVILLE, 2006, p.44).

No desenvolvimento incremental existe um fato interessante, pois não há a necessidade de se utilizar o mesmo processo de desenvolvimento a cada incremento, sendo que, quando as funções presentes em um incremento possuírem especificações detalhadas e bem definidas, o modelo cascata pode ser utilizado especificamente naquele incremento, já que este modelo lida muito bem com requisitos definidos. Mas se a especificação estiver mal-definida, pode-se usar outro modelo de desenvolvimento evolucionário, por ser um modelo mais adequado que o cascata nestas condições, já que não exige definição dos requisitos desde o início, podendo identificar novos requisitos ao longo do processo.

De acordo com Sommerville (2006, p.44), esse processo de desenvolvimento incremental tem diversas vantagens:

• Os clientes não precisam esperar até que todo o sistema seja entregue, para então tirarem proveito dele. O primeiro estágio satisfaz seus requisitos mais importantes e o *software* pode ser imediatamente utilizado.

• Os clientes podem utilizar os primeiros incrementos como um protótipo e obter uma experiência que forneça os requisitos para estágios posteriores do sistema.

• Existe um risco menor de fracasso completo do sistema. Embora possam ser encontrados problemas em alguns incrementos, é provável que alguns incrementos sejam entregues com sucesso ao cliente.

• Como as funções prioritárias são entregues primeiro e incrementos posteriores são integrados a elas, é inevitável que as funções de sistema mais

importantes passem pela maior parte dos testes. Isso significa que é menos provável que os clientes encontrem falhas de *software* na parte mais importante do sistema.

Marcoratti (2004) também apresenta algumas vantagens do processo incremental e iterativo:

- Possibilidade de avaliar mais cedo os riscos e pontos críticos do projeto, e identificar medidas para os eliminar ou controlar;
- Redução dos riscos envolvendo custos a um único incremento. Se a equipe que desenvolve o *software* precisar repetir a iteração, a organização perde somente o esforço mal direcionado de uma iteração, não o valor de um produto inteiro;
- Definição de uma arquitetura que melhor possa orientar todo o desenvolvimento;
- Disponibilização natural de um conjunto de regras para melhor controlar os inevitáveis pedidos de alterações futuras;
- Permite que os vários intervenientes possam trabalhar mais efetivamente pela interação e partilha de comunicação daí resultante.

No entanto, o desenvolvimento incremental também possui alguns problemas ou pontos que devem ser considerados. Incrementos devem ser pequenos, com menos de 20 mil linhas de código, sendo que cada incremento deve realizar pelo menos um serviço e isto pode ser difícil, pois não é simples mapear os requisitos exigidos em incrementos e com o número certo do tamanho. Outro ponto é que os requisitos só são detalhados no momento em que os incrementos são desenvolvidos, sendo difícil identificar facilidades comuns que todos os incrementos tenham como exigência.

De acordo com Wikipedia (2015), os dois padrões mais conhecidos de sistemas iterativos de desenvolvimento são o RUP (Processo Unificado da Rational) e o Desenvolvimento ágil de *software*. Por isso o desenvolvimento iterativo e incremental é também uma parte essencial da Programação Extrema (XP) e outros.

Como sabemos, o desenvolvimento de um produto complexo de *software*, no meio comercial, pode levar meses, em torno de um ano ou até um pouco mais. Assim, dividir o trabalho por meio de iterações e partes menores é mais prático, resultando em incrementos a cada iteração (repetição de uma ou mais ações).

Um dos princípios desta abordagem incremental é permitir que o sistema possa ser refinado e ir ampliando sua qualidade e detalhes por parte da equipe envolvida a cada iteração. Assim, inicialmente, numa primeira iteração, identifica-se uma visão geral e determina a viabilidade econômica do sistema, efetuando grande parte da análise e também uma parte do desenho e implementação. Em seguida, na segunda iteração, após a conclusão da análise, executa-se outra parte do desenho e avança um pouco mais nas implementações. Após a conclusão do desenho, numa terceira iteração, evolui-se a implementação, juntamente com testes e um pouco de integração.

Com estas seguidas iterações, notamos que o amadurecimento do produto de *software* vai evoluindo ao longo do tempo e que a cada iteração é produzido um conjunto de produtos finais, conforme afirma Marcoratti (2014), por meio da realização das seguintes tarefas abaixo e representadas na figura 4.5:

- Análise (refinamento de requisitos, refinamento do modelo conceitual)
- Projeto (refinamento do projeto arquitetural, projeto de baixo nível)
- Implementação (codificação e testes)
- Transição para produto (documentação, instalação, ...)
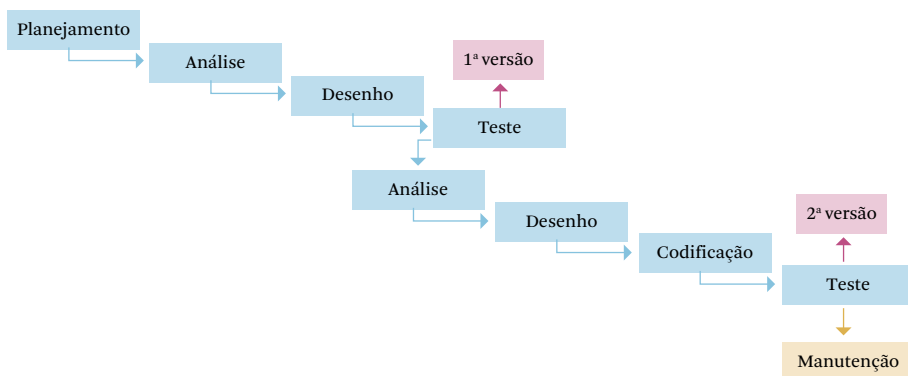
Figura 4.5 – Abordagem incremental. Fonte: MARCORATTI (2014) (Adaptado pelo autor).

Um detalhe importante a ser observado sobre a essência dos processos iterativos é que, conforme Sommerville (2006, p. 43) aponta, "a especificação é desenvolvida em conjunto com o *software*. Contudo, isso entra em conflito com o modelo de suprimentos de muitas organizações, em que a especificação completa do sistema faz parte do contrato" para o desenvolvimento do *software*. Como nesta abordagem incremental e gradativa não há uma especificação completa do sistema, até que o estágio final seja especificado, muitas

vezes isto requer um novo tipo de contrato e aí isto pode ser um entrave para grandes clientes, como órgãos de governo, que geralmente não aceitam este tipo de contrato ou esta abordagem (SOMMERVILLE, 2006, p.43).

4.4 Processo ágil

Uma das principais características dos processos ou metodologias ágeis é a sua natureza adaptativa ou invés de serem preditivas (LIBARDI, 2010, p.2).

O processo ágil apresenta modelos que têm menos ênfase nas definições de atividades e mais ênfase na pragmática e nos fatores humanos do desenvolvimento. Os métodos ágeis, antigamente conhecidos também como **processos leves**, estão em franco desenvolvimento e várias abordagens podem ser encontradas na literatura, conforme apresenta Wazlawick (2013, p. 77).

> Os modelos ágeis de desenvolvimento de *software* seguem uma filosofia diferente da filosofia dos modelos prescritivos. Em vez de apresentar uma "receita de bolo", com fases ou tarefas a serem executadas, eles focam valores humanos e sociais (WAZLAWICK, 2013, p. 77).

Os métodos ágeis possuem foco maior nos resultados do que nos processos, e isto pode torná-los mais leves, no entanto é importante observar que isto não os classifica como um modelo de processos simplistas ou menos complexos.

Os princípios dos modelos ágeis foram claramente colocados no **Manifesto ágil** e assinados por 17 pesquisadores da área, entre os quais Martin Fowler, Alistair Cockburn e Robert Martin. O manifesto estabelece o seguinte (WAZLAWICK, 2013, p. 77):

Nós estamos descobrindo formas melhores de desenvolver *software* fazendo e ajudando outros a fazer. Através desse trabalho chegamos aos seguintes valores:

- Indivíduos e interações estão acima de processos e ferramentas.
- *Software* funcionando está acima de documentação compreensível.
- Colaboração do cliente está acima de negociação de contrato.
- Responder às mudanças está acima de seguir um plano.

Ou seja, enquanto forem valorizados os primeiros, os outros valerão mais.

Análise do Manifesto ágil

"É muito importante entender que os conceitos do manifesto ágil definem preferências e não alternativas no desenvolvimento de *software*, encorajando a focar a atenção em certos conceitos sem eliminar outros. Assim, para seguir os conceitos ágeis, deve-se valorizar mais a certas coisas do que a outras.

1. Indivíduos e interação entre eles mais que processos e ferramentas: os softwares não são construídos por uma única pessoa, eles são construídos por uma equipe, então elas precisam trabalhar juntas (incluindo programadores, testers, projetistas e também o cliente). Processos e ferramentas são importantes, mas não são tão importantes quanto trabalhar juntos.
2. Software em funcionamento mais que documentação abrangente: a documentação deve existir para ajudar pessoas a entender como o sistema foi construído, mas é muito mais fácil entender o funcionamento vendo o sistema funcionar do que através de diagramas que descrevem o funcionamento ou abstraem o uso.
3. Colaboração com o cliente mais que negociação de contratos: somente o cliente pode dizer o que ele espera do software, e normalmente eles não sabem explicar exatamente o que eles esperam, e ainda eles mudam de ideia ao longo do tempo e conforme eles vêm o software funcionando. Ter um contrato é importante para definir as responsabilidades e direitos, mas não deve substituir a comunicação. Trabalhos desenvolvidos com sucesso têm constante comunicação com o cliente para entender suas necessidades e ajudá-lo a descobrir a melhor forma de expressá-las.
4. Responder a mudanças mais que seguir um plano: mudanças é uma realidade no ambiente de negócios e elas acontecem por inúmeras razões: as regras e as leis sofrem alterações, as pessoas mudam de ideia e a tecnologia evolui. O software precisa refletir essas mudanças. Um projeto de software certamente deve ter um plano, mas ele deve ser flexível o suficiente para comportar as mudanças quando elas aparecerem, senão ele se torna irrelevante."

LIBARDI, 2010, p.2.

Para os modelos ágeis, a valorização de processos, ferramentas, documentação, planos e contratos terá mais sentido e mais valor depois que indivíduos, interações, colaborações do cliente, *software* funcionando e resposta às mudanças forem considerados também importantes. De forma, nota-se que processos bem estruturados de nada adiantam se as pessoas não os seguirem, da mesma forma que um *software* bem documentado também não adianta se este não satisfaz os requisitos exigidos pelo cliente ou se estes não funcionam, e assim por diante.

Além dos pontos acima destacados, o manifesto ágil é complementado por doze princípios, citados a seguir (WAZLAWICK, 2013, p. 78):

a) Nossa maior prioridade é satisfazer o cliente através da entrega rápida e contínua de software com valor.

b) Mudanças nos requisitos são bem-vindas, mesmo nas etapas finais do projeto. Processos ágeis usam a mudança como um diferencial competitivo para o cliente.

c) Entregar software frequentemente, com intervalos que variam de duas semanas a dois meses, preferindo o intervalo mais curto.

d) Administradores (*business people*) e desenvolvedores devem trabalhar juntos diariamente durante o desenvolvimento do projeto.

e) Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte e confie que eles farão o trabalho.

f) O meio mais eficiente e efetivo de tratar a comunicação entre/para a equipe de desenvolvimento é a conversa "cara a cara".

g) Software funcionando é medida primordial de progresso.

h) Processos ágeis promovem desenvolvimento sustentado. Os financiadores, usuários e desenvolvedores devem ser capazes de manter o ritmo indefinidamente.

i) Atenção contínua à excelência técnica e bom *design* melhoram a agilidade.

j) Simplicidade – a arte de maximizar a quantidade de um trabalho não feito – é essencial.

k) As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizadas.

l) Em intervalos regulares, a equipe reflete sobre como se tornar mais efetiva e então ajusta seu comportamento de acordo com essa meta.

Embora exista um número significativo de modelos atuais considerados ágeis e diferentes entre si, praticamente todos eles adotam estes princípios relacionados como pontos fundamentais em seu funcionamento.

De acordo com Pressman (2011, p. 82), os métodos ágeis, em essência, se desenvolveram em um esforço para eliminar fraquezas reais e perceptíveis da engenharia de *software* convencional. Mesmo oferecendo benefícios importantes, vale ressaltar que o desenvolvimento ágil não é indicado para todos os projetos, produtos, pessoas e situações. Este método também pode ser aplicado como uma filosofia geral para todos os trabalhos de *software*.

Nos tempos de uma economia moderna, é muito difícil prever como um sistema computacional irá evoluir com o tempo, como, por exemplo, as aplicações baseadas na Web. Vários fatores influenciam nestas mudanças e as condições de mercado também mudam rapidamente, bem como as necessidades dos usuários finais e outros fatores relacionados às ameaças competitivas no mercado. Muitas vezes não será possível definir completamente requisitos antes que se inicie o projeto, e então será necessário ser ágil o suficiente para dar uma resposta ao meio onde o negócio está inserido, deixando-o fluir de acordo com as necessidades (PRESSMAN, 2011, p. 82).

Esta fluidez está associada a mudanças, e na maioria das vezes isto é caro, principalmente se estas mudanças forem isentas de controle e gestão adequada. A habilidade de reduzir custos das mudanças é uma das características mais convincentes da abordagem ágil, ao longo de todo o processo de *software* (PRESSMAN, 2011, p. 82).

Sendo assim, quer dizer que os valiosos princípios da engenharia de *software*, conceitos, métodos, processos e ferramentas podem ser descartados? É evidente que não! Este reconhecimento e diferenças dos processos ágeis podem vir a contribuir com a disciplina de Engenharia de *Software*, permitindo sofisticá-la ainda mais e adaptá-la facilmente aos desafios que a demanda por agilidade tem apresentado.

O que é agilidade?

No contexto da engenharia de *software*, o que é agilidade?

Atualmente, agilidade tornou-se a palavra da moda quando se descreve um moderno processo de *software*. Todo mundo é ágil. Uma equipe ágil é aquela rápida e capaz de responder apropriadamente a mudanças. Mudanças têm muito a ver com desenvolvimento de *software*. Mudanças no *software* que está sendo criado, mudanças nos membros da equipe, mudanças devido a novas tecnologias, mudanças de todos os tipos que poderão ter um impacto no produto que está em construção ou no projeto que cria o produto. Suporte para mudanças deve ser incorporado em tudo o que fazemos em *software*, algo que abraçamos porque é o coração e a alma do *software*. Uma equipe

ágil reconhece que o *software* é desenvolvido por indivíduos trabalhando em equipes e que as habilidades dessas pessoas, suas capacidades em colaborar estão no cerne do sucesso do projeto.

Agilidade consiste em algo mais que uma resposta à mudança, abrangendo a filosofia proposta no manifesto citado no início deste capítulo. Ela incentiva a estruturação e as atitudes em equipe que tornam a comunicação mais fácil (entre membros da equipe, entre o pessoal ligado à tecnologia e o pessoal da área comercial, entre os engenheiros de *software* e seus gerentes). Enfatiza a entrega rápida do *software* operacional e diminui a importância dos artefatos intermediários (nem sempre um bom negócio); assume o cliente como parte da equipe de desenvolvimento e trabalha para eliminar a atitude de "nós e eles", que continua a invadir muitos projetos de *software*; reconhece que o planejamento em um mundo incerto tem seus limites e que o plano (roteiro) de projeto deve ser flexível.

A agilidade pode ser aplicada a qualquer processo de *software*. Entretanto, para obtê-la é essencial que seja projetado para que a equipe possa adaptar e alinhar (racionalizar) tarefas; possa conduzir o planejamento compreendendo a fluidez de uma abordagem do desenvolvimento ágil; possa eliminar tudo, exceto os artefatos essenciais, conservando-os enxutos; e enfatize a estratégia de entrega incremental, conseguindo entregar ao cliente, o mais rapidamente possível, o *software* operacional para o tipo de produto e ambiente operacional.

PRESSMAN, 2011, p. 82

Ainda de acordo com Pressman, (2011, p.82), qualquer processo ágil de *software* é caracterizado de uma forma que se relacione a uma série de preceitos-chave acerca da maioria dos projetos de *software*:

1. É difícil afirmar antecipadamente quais requisitos de *software* irão persistir e quais sofrerão alterações. É igualmente difícil prever de que maneira as prioridades do cliente sofrerão alterações conforme o projeto avança.
2. Para muitos tipos de *software*, o projeto e a construção são "interconduzidos" . Ou seja, ambas as atividades devem ser realizadas em sequência (uma atrás da outra), para que os modelos de projeto sejam provados conforme sejam criados. É difícil prever quanto de trabalho de projeto será necessário antes que a sua construção (desenvolvimento) seja implementada para avaliar o projeto.

3. Análise, projeto, construção (desenvolvimento) e testes não são tão previsíveis (do ponto de vista de planejamento) quanto gostaríamos que fosse.

Esses três pontos levantam questionamentos bem pertinentes: como administrar a imprevisibilidade por meio de processos? Como estes processos poderiam ser criados? A adaptação destes processos é o caminho a ser percorrido, portanto, um processo ágil deve ser adaptável. Estas adaptações podem ocorrer de maneira incremental e para que isto aconteça, é necessário colher *feedbacks* do cliente, por parte das equipes ágeis.

Ao longo da história da engenharia de *software*, surgiram dezenas de metodologias e descrições de processos, métodos e notações de modelagem, ferramentas e tecnologias obsoletas. Algumas tiveram maior adesão que outras e outras foram sendo ofuscadas pelo surgimento de novas entrantes, supostamente melhores. Com o surgimento de vários modelos de processos ágeis, a história seguindo o mesmo caminho que antes, onde notamos disputas pela aceitação das novas metodologias ou daquelas que se concretizaram na opinião da comunidade de desenvolvimento de *software*.

Dentre estes modelos, o Extreme Programming (XP) é o mais amplamente utilizado. No entanto, muitos outros têm sido propostos e encontram-se em uso no setor, conforme relaciona Pressmann (2011, p. 93), apontando os mais comuns:

- Desenvolvimento de *software* adaptativo (*Adaptive Software Development*, ASD)
- Scrum
- Método de desenvolvimento de sistemas dinâmicos (*Dynamic Systems Development Method*, DSDM)
- Crystal
- Desenvolvimento dirigido a Funcionalidades (*Feature Drive Development*, FDD)
- Desenvolvimento de *software* enxuto (*Lean Software Development*, LSD)
- Modelagem ágil (*Agile Modeling*, AM)
- Processo unificado ágil (*Agile Unified Process*, AUP)

4.4.1 XP – eXtreme Programming (Programação Extrema)

O modelo ágil XP (*eXtreme Programming*) surgiu no final da década de 1990, nos Estados Unidos, inicialmente adequado para pequenas e médias equipes, baseado em uma série de valores, princípios e regras (WAZLAWICK, 2013, p. 98), com desenvolvimento de *software* contendo requisitos vagos e com rápidas modificações (SOARES, 2014, p. 3).

Os principais valores do XP são: simplicidade, respeito, comunicação, *feedback* e coragem. Wazlawick (2013, p. 98) descreve estes valores:

SIMPLICIDADE	segundo o Chaos Report (Standish Group, 1995),mais da metade das funcionalidades introduzidas em sistemas nunca é usada. XP sugere como valor a simplicidade, ou seja, a equipe deve se concentrar nas funcionalidades efetivamente necessárias, e não naquelas que poderiam ser necessárias, mas de cuja necessidade real ainda não se tem evidência.
RESPEITO	respeito entre os membros da equipe, assim como entre a equipe e o cliente, é um valor dos mais básicos, que dá sustentação a todos os outros. Se não houver respeito, a comunicação falha e o projeto afunda.
COMUNICAÇÃO	em desenvolvimento de *software*, a comunicação é essencial para que o cliente consiga dizer aquilo de que realmente precisa. O XP preconiza comunicação de boa qualidade, preferindo encontros presenciais em vez de videoconferências, videoconferências em vez de telefonemas, telefonemas em vez de e-mails, e assim por diante. Ou seja, quanto mais pessoal e expressiva a forma de comunicação, melhor.
FEEDBACK	o projeto de *software* é reconhecidamente um empreendimento de alto risco. Cientes disso, os desenvolvedores devem buscar obter *feedback* o quanto antes para que eventuais falhas de comunicação sejam corrigidas o mais rapidamente possível, antes que os danos se alastrem e o custo da correção seja alto.
CORAGEM	pode-se dizer que a única coisa constante no projeto de um *software* é a necessidade de mudança. Para os desenvolvedores XP, é necessário confiar nos mecanismos de gerenciamento da mudança para ter coragem de abraçar as inevitáveis modificações em vez de simplesmente ignorá-las por estarem fora do contrato formal ou por serem muito difíceis de acomodar.

Uma série de princípios básicos são definidos a partir desses valores:

- *Feedback* rápido.
- Presumir simplicidade.
- Mudanças incrementais.
- Abraçar mudanças.
- Trabalho de alta qualidade.

A XP preconiza mudanças incrementais e *feedback* rápido, além de considerar a mudança algo positivo, que deve ser entendido como parte do processo desde o início. Além disso, a XP valoriza o aspecto da qualidade, pois considera que pequenos ganhos a curto prazo pelo sacrifício da qualidade não são compensados pelas perdas a médio e a longo prazo (WAZLAWICK, 2013, p. 99).
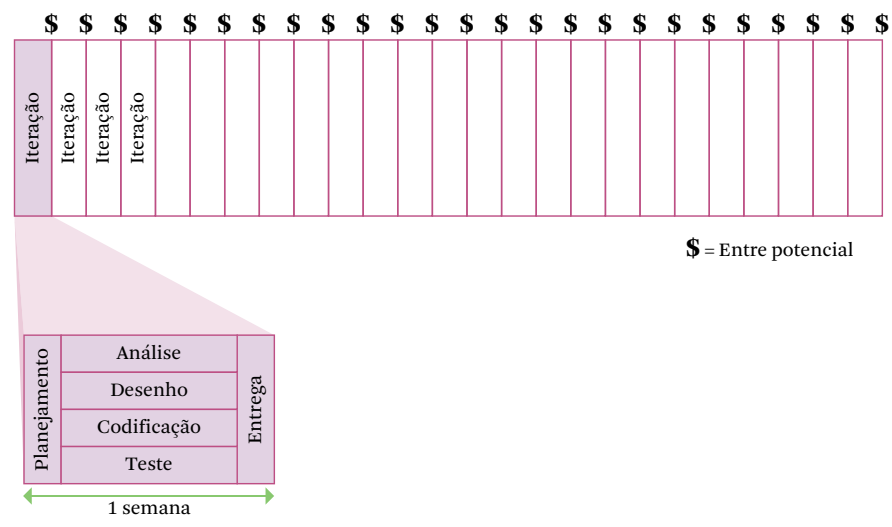
Figura 4.6 – Ciclo de vida XP. Fonte: Shore e Warden (2008, p. 35) (Adaptado pelo autor).

Dentre as características desta abordagem, a **priorização de funcionalidades mais importantes** proporciona entregas parciais daquilo que é mais importante, mesmo que todo o trabalho do projeto ainda não tenha sido concluído.

De acordo com Soares (2014, p. 3), o primeiro projeto a usar XP foi o C3, da Chrysler. Após anos de fracasso utilizando metodologias tradicionais, com o uso da XP o projeto ficou pronto em pouco mais de um ano.

A XP baseia-se nas 12 práticas apresentadas por Beck (1999) a seguir (SOARES, 2014, p. 4):

1. **Planejamento (Jogo de planejamento):** consiste em decidir o que é necessário ser feito e o que pode ser adiado no projeto. A XP baseia-se em requisitos atuais para desenvolvimento de *software*, não em requisitos futuros. Além disso, a XP procura evitar os problemas de relacionamento entre a área de negócios (clientes) e a área de desenvolvimento. As duas áreas de vem cooperar para o sucesso do projeto e cada uma deve focar em partes específicas do projeto.

Desta forma, enquanto a área de negócios deve decidir sobre o escopo, a composição das versões e as datas de entrega, os desenvolvedores devem decidir sobre as estimativas de prazo, o processo de desenvolvimento e o cronograma detalhado para que o *software* seja entregue nas datas especificadas.

2. **Entregas freqüentes:** visa à construção de um *software* simples, e conforme os requisitos surgem, há a atualização do *software*. Cada versão entregue deve ter o menor tamanho possível, contendo os requisitos de maior valor para o negócio. Idealmente devem ser entregues versões a cada mês, ou no máximo a cada dois meses, aumentando a possibilidade de *feedback* rápido do cliente. Isto evita surpresas caso o *software* seja entregue após muito tempo e melhora as avaliações do cliente, aumentando a probabilidade do *software* final estar de acordo com os requisitos do cliente.

3. **Metáfora:** são as descrições de um *software* sem a utilização de termos técnicos, com o intuito de guiar o desenvolvimento do *software*.

4. **Projeto simples:** o programa desenvolvido pelo método XP deve ser o mais simples possível e satisfazer os requisitos atuais, sem a preocupação de requisitos futuros. Eventuais requisitos futuros devem ser adicionados assim que eles realmente existirem.

5. **Testes:** a XP focaliza a validação do projeto durante todo o processo de desenvolvimento. Os programadores desenvolvem o *software* criando primeiramente os testes.

6. **Programação em pares:** a implementação do código é feita em dupla, ou seja, dois desenvolvedores trabalham em um único computador. O desenvolve dor que está com o controle do teclado e do *mouse* implementa o código, enquanto o outro observa continuamente o trabalho que está sendo feito, procurando identificar erros sintáticos e semânticos e pensando estrategicamente em como melhorar o código que está sendo implementado. Esses papéis podem e devem ser alterados continuamente. Uma grande vantagem da programação em dupla é a possibilidade de os desenvolvedores estarem continuamente aprendendo um com o outro.

7. **Refatoração:** focaliza o aperfeiçoamento do projeto do *software* e está presente em todo o desenvolvimento. A refatoração deve ser feita apenas quando é necessário, ou seja, quando um desenvolvedor da dupla, ou os dois, percebe que é possível simplificar o módulo atual sem perder nenhuma funcionalidade.

8. Propriedade coletiva: o código do projeto pertence a todos os membros da equipe. Isto significa que qualquer pessoa que percebe que pode adicionar valor a um código, mesmo que ele próprio não o tenha desenvolvido, pode fazê-lo, desde que faça a bateria de testes necessária. Isto é possível porque na XP todos são responsáveis pelo *software* inteiro. Uma grande vantagem desta prática é que, caso um membro da equipe deixe o projeto antes do fim, a equipe consegue continuar o projeto com poucas dificuldades, pois todos conhecem todas as partes do *software*, mesmo que não seja de forma detalhada.

9. Integração contínua: é a prática de interagir e construir o sistema de *software* várias vezes por dia, mantendo os programadores em sintonia, além de possibilitar processos rápidos. Integrar apenas um conjunto de modificações de cada vez é uma prática que funciona bem porque fica óbvio quem deve fazer as correções quando os testes falham: a última equipe que integrou código novo ao *software*. Esta prática é facilitada com o uso de apenas uma máquina de integração, que deve ter livre acesso a todos os membros da equipe.

10. 40 horas de trabalho semanal: a XP assume que não se deve fazer horas extras constantemente. Caso seja necessário trabalhar mais de 40 horas pela segunda semana consecutiva, existe um problema sério no projeto que deve ser resolvido não com aumento de horas trabalhadas, mas com melhor planejamento, por exemplo. Esta prática procura ratificar o foco nas pessoas e não em processos e planejamentos. Caso seja necessário, os planos devem ser alterados, ao invés de sobrecarregar as pessoas.

11. Cliente presente: é fundamental a participação do cliente durante todo o desenvolvimento do projeto. O cliente deve estar sempre disponível para sanar todas as dúvidas de requisitos, evitando atrasos e até mesmo construções erradas. Uma ideia interessante é manter o cliente como parte integrante da equipe de desenvolvimento.

12. Código padrão: padronização na arquitetura do código, para que este possa ser compartilhado entre todos os programadores.

De acordo com Prikladnicki et al. (2014, p. 38), a XP possui uma proposta de não ser uma metodologia radicalmente técnica e imediatista. A proposta de usar ao máximo as boas práticas de engenharia de *software* já adotadas e reconhecidas pela indústria é que justifica o extremismo. Há uma significativa sinergia no conjunto das práticas existentes e isto representa um grande ponto para a XP.

A XP é muito indicada para ser utilizada em projetos em que os envolvidos (*stakeholders*) não possuem os requisitos estáveis, com incertezas quanto ao que desejam e com oscilações de opiniões ao longo do processo de desenvolvimento de *software*. O contato direto com o cliente, um retorno eficiente por meio de *feedbacks*, são meios que a equipe de desenvolvimento dispõem para adaptar às mudanças que ocorrem rapidamente neste cenário. Nas metodologias tradicionais isto pode ser um sério problema, já que todos os requisitos devem ser muito bem definidos desde o princípio.

Por fim, a sua forte ligação e apelo com a engenharia de *software* faz com que a XP tenha a atenção e atração, cada vez mais, de desenvolvedores de *softwares*. A metodologia proposta faz uma grande aproximação entre o cliente e a equipe desenvolvedora, garantindo maior proximidade e integração no processo de desenvolvimento do produto de *software*, e também compartilhar trocas de conhecimentos e estabelecer relações mais produtivas na equipe, principalmente por conta da programação e pares.

4.4.2 Scrum

O Scrum é um *framework* ágil utilizado para auxiliar o gerenciamento de projetos considerados complexos, bem como no desenvolvimento de produtos, de maneira incremental e iterativa. É muito utilizado em desenvolvimento de *softwares* por ter um conjunto de práticas leves e objetivas (PRIKLADNICKI et al., 2014, p. 22).

Diferente da grande maioria das outras metodologias que abordam o desenvolvimento a partir das tarefas de programação de modo explícito, o Scrum é voltado para o gerenciamento dos projetos de *software*. O seu uso em conjunto com outras metologias, como o XP, por exemplo, proporcionam resultados bem interessantes, conforme afirma Oliveira (2003, p. 24). O fato de ser um *framework*, sendo um conjunto de práticas estruturadas e sistematizadas, facilita esta associação e complemento a outras metodologias.

A concepção inicial do Scrum deu-se na indústria automobilística , e o modelo pode ser adaptado a diferentes áreas da produção de *software* (WAZLAWICK, 2013, p. 92). O seu nome foi inspirado a partir de uma jogada existente no *rugby*, chamada "Scrum" e, portanto, não é uma sigla. É comum ainda encontrarmos por aí a associação de Scrum como uma metodologia de desenvolvimento

de *software*, no entanto, vale ressaltar que o mesmo é um *framework* que pode ser utilizado à uma metodologia e não necessariamente uma metodologia própria. Portanto, ao vermos a relação de métodos ágeis e Scrum, devemos ficar atentos a este detalhe. De fato o Scrum é utilizado nos processos ágeis de desenvolvimento, mas como um conjunto de prática associada a uma metodologia.

Libardi (2010, p.5) destaca um ponto importante sobre o Scrum, onde ressalta que o mesmo trabalha com a complexidade de desenvolvimento de *softwares* por meio do controle de inspeção, adaptação e visibilidade de requisitos de um processo empírico, fazendo uso de uma série de regras e práticas, onde controle não significa controle para criar o que foi previsto, e sim controlar o processo para orientar o trabalho em direção a um produto com o maior valor agregado possível.

A estrutura iterativa e incremental é empregada para atingir estes objetivos no Scrum, basicamente do seguinte modo:

- A equipe analisa o que deverá ser feito no início de cada iteração;
- após a análise, faz-se uma seleção daquilo que poderia ser um incremento de valor ao produto final da iteração;
- a equipe se empenha ao máximo para desenvolver aquilo que foi selecionado naquela iteração;
- por fim, apresenta-se o incremento de funcionalidade desenvolvido para que os envolvidos possam avaliar e solicitar novas alterações no momento oportuno.

Dessa forma, notamos que o núcleo principal do Scrum é a iteração. A equipe analisa a fundo os requisitos, as tecnologias e habilidades envolvidas, a cada iteração, e então dividem-se para desenvolver e entregar o melhor *software* possível, fazendo ainda as adaptações necessárias diariamente, de acordo com o aparecimento de complexidades, dificuldades ou até mesmo surpresas (LIBARDI, 2010, p.5).

O trabalho no Scrum é estruturado em ciclos de desenvolvimento (geralmente vai de duas semanas a um mês), chamados de *Sprints*, e em cada ciclo (*sprint*) são priorizadas as atividades que serão realizadas neste trabalho, a partir de lista de requisitos, conhecida como *Backlog* do Produto *(Product Backlog)*.

O Scrum possui características que podem ser citadas, de acordo com Prikladnicki et al. (2014, p. 23):

• as equipes pequenas e multidisciplinares que trabalham de forma integrada em um ambiente aberto para produzir versões incrementais de um produto de *software* em iterações curtas;
• as equipes se auto-organizam para planejar e desenvolver o trabalho das *sprints*, ou seja, a liderança para fazer esse trabalho é diluída em cada integrante da equipe;
• o trabalho em equipe é facilitado pelo *ScrumMaster*, que não trabalha diretamente com atividades técnicas, mas remove impedimentos e reforça os pontos centrais do Scrum ao longo do desenvolvimento do produto;
• o trabalho é organizado a partir do *Backlog do Produto*, constantemente revisado e priorizado;
• a comunicação e cooperação entre as equipes se intensificam ao longo do desenvolvimento das funcionalidades do produto.

Existem papéis ou agentes responsáveis pelas atividades em um projeto Scrum e, diferentemente de como ocorre em outros processos, em que contam com a presença de um responsável por todo o projeto, no Scrum há uma distribuição da gestão do projeto ou do produto. Esta gestão é dividida entre os papéis: Dono do Produto (*Product Owner*), *ScrumMaster* e Equipe *Scrum* (ou Equipe de desenvolvimento). A descrição destes papéis pode ser assim compreendida, de acordo com Silva (2014) e Prikladnicki et al. (2014, p. 27):

• **Product Owner (Dono do Produto)** – é o representante do cliente dentro da equipe. Responsável por controlar e gerenciar o *Product Backlog* (lista de requisitos que compõem o produto final) e efetuar o aceite das entregas de cada ciclo iterativo de desenvolvimento, além de garantir o retorno sobre o investimento, definir a visão do produto, gerenciar riscos e avaliar, aceitar ou rejeitar o que será entregue no final de cada iteração;
• **ScrumMaster** – além de ser a pessoa que mais conhece Scrum, em relação aos outros papéis, é o líder e facilitador da equipe. Responsável por remover impedimentos que possam atrapalhar a produtividade da equipe, auxilia o *Product Owner* na elaboração do Product Backlog e garante o respeito às práticas do Scrum. É também o responsável por fazer as reuniões fluírem, de forma adequada, sendo um facilitador, embora não seja o responsável por conduzi-las;

- **Equipe Scrum** – é responsável pelo desenvolvimento dos incrementos do produto, com o compromisso de entregá-los ao final de cada iteração. São colaboradores multifuncionais e autogerenciáveis responsáveis pela construção do produto.

No Scrum existem algumas **cerimônias ou eventos** que possuem duração fixa e que são realizados em intervalos regulares, sendo um momento oportuno para inspecionar e adaptar as atividades de desenvolvimento do produto. Estas cerimônias ou eventos podem ser divididos em: *Planning Meeting*, *Daily Scrum*, *Sprint Review Meeting* e Retrospectiva da *Sprint*.

Em cada *Sprint* é realizado o *Planning Meeting* (Reunião de Planejamento da Sprint), em seu início. O *Product Owner* levanta as prioridades a partir do *Product Backlog* e as coloca no topo, de acordo com a capacidade que a equipe de desenvolvimento for capaz de cumprir naquela *sprint*. A quantidade de itens é então avaliada pela equipe e é ela que avalia e determina esta quantidade e define a meta da *sprint*, servindo como um roteiro do que deverá ser trabalhado no desenvolvimento durante a *sprint*. Estes itens presentes e organizados em prioridades no *backlog* são chamados de *Stories*. Estas *stories* representam as estórias dos usuários, de acordo com as solicitações ou necessidades dos usuários.

Durante a reunião de planejamento da *sprint*, duas perguntas importantes são realizadas: (a) o que será entregue no incremento resultante nesta *sprint*? (b) como faremos para entregar o incremento nesta *sprint*? (PRIKLADNICKI et al., 2014, p. 22).

Diariamente, no início do dia, a equipe realiza uma reunião conhecida como Daily Scrum (Scrum diária), com o propósito de compartilhar conhecimento sobre alguns pontos significativos, como o que foi feito no dia anterior, identificar dificuldades ou impedimentos e relacionar as prioridades das tarefas do dia (SILVA, 2014). Esta reunião tem um tempo limitado a 15 minutos e durante este tempo os membros respondem entre eles à três perguntas: (a) o que foi feito desde ontem (ou última *Daily Scrum*); (b) o que pretende ser feito até a próxima reunião diária (*Daily Scrum*) e (c) há algo que esteja impedindo a conclusão da tarefa? Nesta reunião, a troca de informações e esta disseminação do conhecimento ficam voltadas entre os membros da equipe, sem necessidade de gerar relatórios aos demais papéis. Em casos de problemas ou dificuldades, a própria equipe se auto-organiza para tentar saná-los. Caso seja algo que a equipe não consiga ou não tenha condições de tratar, sem condições de resolver, esta

atividade é classificada como Impedimento e é passada ao ScrumMaster para que este solucione o caso ou busque meios para tal.

Quando a *Sprint* chega ao seu final acontece a revisão da *sprint*, por meio da reunião chamada de ***Sprint Review Meeting***, efetuando-se a entrega parcial e o planejamento do novo ciclo. Além da equipe Scrum, participam desta reunião todos aqueles que estiverem interessados no produto. O principal objetivo desta reunião é avaliar o que foi desenvolvido pela equipe ao longo da sprint e obter *feedback* de todos os presentes, para que se possa planejar e adequar a próxima *sprint*.

Há ainda um último evento de uma *Sprint*, chamado de **Retrospectiva da *Sprint***. Nele, os membros da equipe se reúnem logo após a reunião de revisão da *sprint* e focam nas melhorias dos processos, práticas utilizadas, ferramentas e trocas de experiências que tiveram ao longo daquela *Sprint*, compartilhando, entre eles, o que poderia ser melhorado para ser aplicado na próxima *sprint*, identificando ainda as principais medidas tomadas para a solução de problemas.

A estrutura e o fluxo do Scrum que apresentam esta dinâmica estão representados na figura 4.7, abaixo:
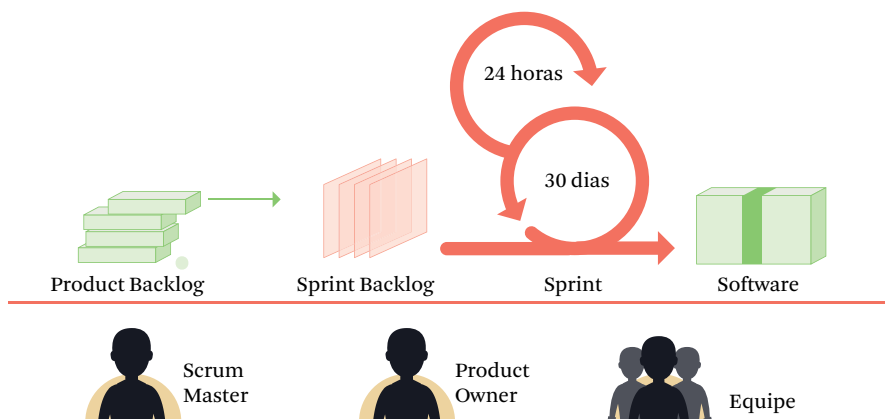
Figura 4.7 – Estrutura do Scrum. Wikimedia. Disponível em: <http://upload.wikimedia.org/wikipedia/commons/5/58/Scrum_process.svg>.

Durante a *sprint*, cabe ao *product owner* manter as *stories* atualizadas, indicando as tarefas já realizadas e aquelas que ainda estão por concluir. Recomenda-se que estas atividades estejam claramente exibidas em um gráfico atualizado diariamente e à vista de todos. Um exemplo de quadro de andamento de atividades é apresentado na figura 4.8 (WAZLAWICK, 2013, p. 94).
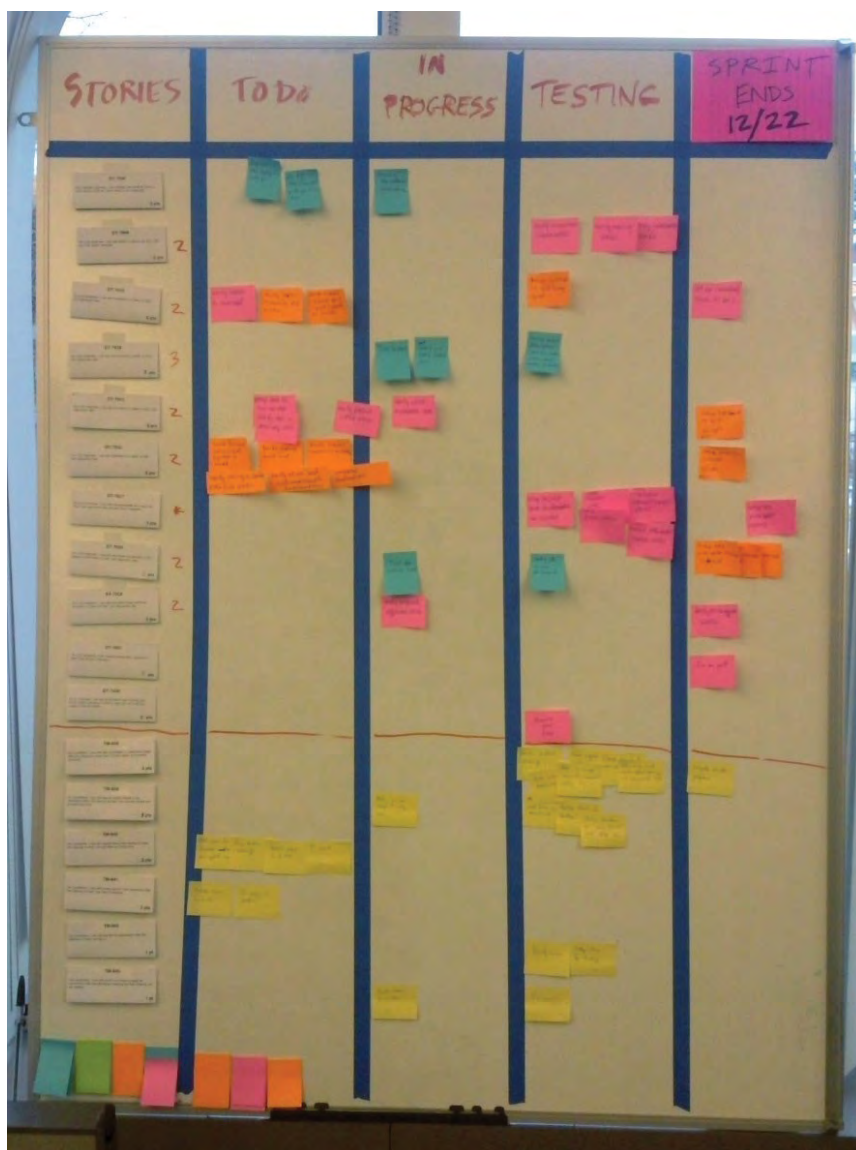
Figura 4.8 – Quadro de acompanhamento de uma *Sprint*, também conhecido por Kanban. Wikimedia. Disponível em: <http://commons.wikimedia.org/wiki/File:Scrum_task_board.jpg>

Este quadro de acompanhamento de uma *Sprint* também é conhecido por *Kanban*. Nele, a primeira coluna representa as tarefas (ou também as *Stories*) definidas para uma determinada *Sprint* e as outras colunas são as fases de desenvolvimento por onde passam essas tarefas (SILVA, 2014).

Uma outra forma de monitoramento bastante utilizada para acompanhar o progresso do projeto é o uso do gráfico *Burndown* (figura 4.9). Por meio dele é possível verificar a quantidade de trabalho restante, sendo ótimo para correlacionar a quantidade de trabalho que falta ser feita e o progresso da equipe, podendo ser checado em qualquer ponto (LIBARDI, 2010, p. 13). No gráfico Burndown, há basicamente as linhas que mostram as tarefas remanescentes, as que foram completadas e a linha ideal que apresenta a data estimada da conclusão final.
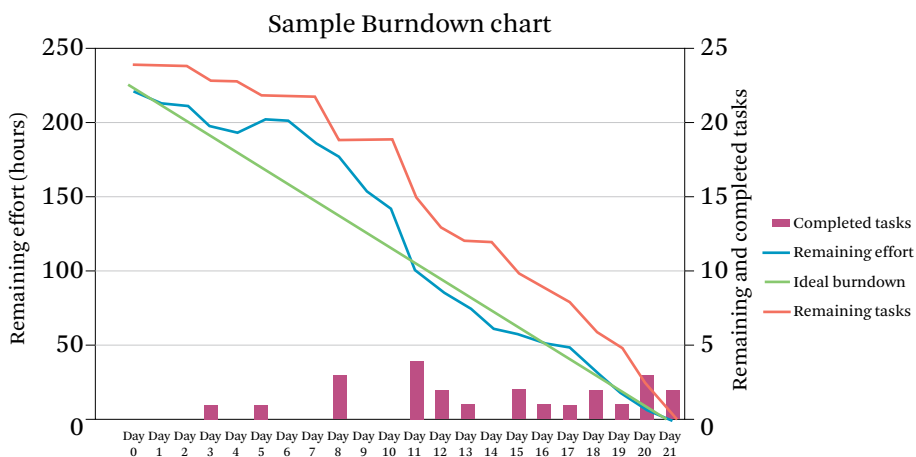
Figura 4.9 – Exemplo de um gráfico Burndown. Pablo Straub. Wikimedia (http://en.wikipedia.org/wiki/Scrum_%28software_development%29#/media/File:SampleBurndownChart.png)

Comparação entre metodologias

"A maioria das metodologias ágeis nada possuem de novo. O que as diferencia das metodologias tradicionais são o enfoque e os valores. A ideia das metodologias ágeis é o enfoque nas pessoas e não em processos ou algoritmos. Além disso, existe a preocupação de gastar menos tempo com documentação e mais com a implementação. De certa forma, apesar da XP ser uma metodologia nova e ser considerada por muitos como uma revolução, ela não apresenta muitos pontos revolucionários. Na verdade, a XP agrupa uma série de práticas que têm sido usadas desde o início da computação eletrônica, como a programação em duplas e a propriedade coletiva do código.

Uma característica das metodologias ágeis é que elas são adaptativas ao invés de serem preditivas. Com isso, elas se adaptam a novos fatores decorrentes do desenvolvimento do projeto, ao invés de procurar analisar previamente tudo o que pode acontecer

no decorrer do desenvolvimento. Essa análise prévia é difícil e apresenta alto custo, além de tornar-se um problema caso não se queira fazer alterações nos planejamentos. Por exemplo, para seguir estritamente o planejamento, pode ser necessário que a equipe trabalhe sobre pressão e faça muitas horas extras, o que prejudica a qualidade do *software*.

Para ser realmente considerada ágil a metodologia deve aceitar a mudança ao invés de tentar prever o futuro. O problema não é a mudança em si, mesmo porque ela ocorrerá de qualquer forma. O problema é como receber, avaliar e responder às mudanças. Como exemplo, as aplicações baseadas em Web são melhor modeladas usando metodologias ágeis, uma vez que o ambiente Web é muito dinâmico.

As metodologias ágeis ainda estão em sua infância, mas já apresentam resultados efetivos. Por exemplo, um artigo de Charette, R., (2001) comparando métodos ágeis com as metodologias tradicionais pesadas mostrou que os projetos usando os métodos ágeis obtiveram melhores resultados em termos de cumprimento de prazos, de custos e padrões de qualidade. Além disso, o mesmo estudo mostra que o tamanho dos projetos e das equipes que utilizam as metodologias ágeis tem crescido. Apesar de serem propostas idealmente para serem utilizadas por equipes pequenas e médias (até 12 desenvolvedores), aproximadamente 15% dos projetos que usam metodologias ágeis estão sendo desenvolvidos por equipes de 21 a 50 pessoas, e 10% dos projetos são desenvolvidos por equipes com mais de 50 pessoas, considerando um universo de 200 empresas usado no estudo".

SOARES, 2014, p. 5.

ATIVIDADES

01. Explique o que é e para que serve um ciclo de vida do *software*?

02. Pesquise sobre o manifesto ágil e descreva 5 principais características das metodologias ágeis.

03. Considerando que um projeto de desenvolvimento de *software* esteja utilizando o modelo cascata como metodologia escolhida, comente como este projeto reagiria com a indefinição de requisitos e mudanças ao longo do processo de desenvolvimento.

04. Sobre o modelo cascata ou clássico, é **incorreto** afirmar que:

a) comparado com outros modelos de desenvolvimento de software, ele é mais rígido e menos administrativo.
b) o modelo cascata é um dos modelos mais importantes, pois é referência para muitos outros modelos, embora seja criticado por ser linear, rígido e monolítico.
c) este modelo argumenta que cada atividade apenas deve ser iniciada quando a outra estiver terminada e verificada.
d) é derivado das práticas de engenharia tradicional, desta forma existe um estabelecimento de uma ordem no desenvolvimento de grandes produtos de software.
e) este modelo é orientado para uma documentação superficial de cada etapa, sem muitos detalhes.

05. Desenvolver *software* é uma atividade difícil e arriscada. Segundo as estatísticas, entre os maiores riscos estão: gastos que superam o orçamento, consumo de tempo que supera o cronograma, funcionalidades que não resolvem os problemas dos usuários, baixa qualidade dos sistemas desenvolvidos e cancelamento do projeto por inviabilidade. O desenvolvimento ágil de *software* fundamenta-se no Manifesto Ágil. Segundo ele, deve-se valorizar:

a) seguimento de um plano em vez de resposta à mudança.
b) mudança de respostas em vez do seguimento de um plano.
c) indivíduos e intenções junto a processos e ferramentas.
d) documentação extensiva operacional em vez de software funcional.
e) indivíduos e interações em vez de processos e ferramentas.

REFLEXÃO

Afinal, qual seria o melhor método de desenvolvimento ou modelo de ciclo de vida de *software*?

Ao estudar este capítulo, eventualmente você pode ter questionado esta pergunta. Quando lidamos com diferentes paradigmas, ou diferentes formas de fazer valer cumprir o mesmo objetivo, que é o de desenvolver *softwares* de qualidade, podemos nos encontrar em situações de dúvidas sobre qual caminho seguir, em relação às diferentes abordagens e metologias existentes. Há uma série de fatores que devem ser analisados e esta não é uma resposta simples, direta e tão objetiva. Estes fatores podem estar relacionados com a qualidade dos requisitos de sistema, grau de maturidade da equipe, tamanho da equipe, grau do envolvimento e acesso ao cliente, entre outros.

Deste modo, podemos observar que cada metodologia, cada modelo de ciclo de vida de *software* possui suas peculiaridades e capacidades, e podemos e devemos fazer alguns questionamentos, como, por exemplo, se o projeto de *software* em questão trabalha com a compreensão deficiente dos requisitos; trabalha com a compreensão deficiente da qualidade; produz sistemas de alta confiança; será fácil modificar o sistema em versões futuras; gerencia riscos; pode ser alimentado em um cronograma predefinido; permite correções no meio do projeto; possibilita ao cliente visibilidade do progresso; possibilita ao cliente visibilidade do progresso do gerenciamento; requer pouca gerência ou experiência para usá-lo.

Com estes e outros questionamentos, podemos analisar alguns fatores que poderão contribuir para a escolha do método de desenvolvimento que melhor se adequará à equipe de desenvolvimento e a todos os envolvidos neste processo.

LEITURA

Existem várias recomendações para os tópicos apresentados neste capítulo.

Como publicação impressa, o livro de Roger Pressman, 2011, é fundamental e uma excelente referência para a área, assim como os livros de Shari L. Pfleeger.

- PFLEEGER, S. L. **Engenharia de *Software*** - Teoria e Prática. São Paulo: Prentice Hall, 2010
- PRESSMAN, R. **Engenharia de *Software***. São Paulo: McGraw-Hill, 2011.

Em seu livro, Pressman ainda sugere livros como os de Shaw e Warden (*The Art of Agile Development*, O'Reilly Media, Inc., 2008), Hunt *(Agile Software Building*, Springer, 2005) e Carmichael e Haywood (*Better Software Faster*, Prentice-Hall, 2002), que trazem discussões interessantes sobre o tema. Aguanno (*Managing Agile Projects, Multi-Media Publications*, 2005), Highsmith (*Agile Project Management: Creating Innovative Products,* Addison-Wesley, 2004) e Larman (*Agile and Iterative Development: A Manager's Guide*, Addison-Wesley, 2003) apresentam uma visão geral sobre gerenciamento e consideram as questões envolvidas no gerenciamento de projetos. Highsmith (*Agile Software Development Ecosystems*, Addison-Wesley, 2002) retrata uma pesquisa de princípios, processos e práticas ágeis. Uma discussão que vale a pena sobre o delicado equilíbrio entre agilidade e disciplina é fornecida por Booch e seus colegas (*Balancing Agility and Discipline*, Addison-Wesley, 2004).

REFERÊNCIAS BIBLIOGRÁFICAS

BECK, K. **Programação Extrema Explicada**. Bookman, 1999. p. 182.

CHARETTE, R. "**Fair Fight? Agile Versus Heavy Methodologies**", Cutter Consor- tium E-project Management, Advisory Service, 2, 13, 2001.

LIBARDI, P. L. O; BARBOSA, V. **Métodos Ágeis. Faculdade de Tecnologia** - FT, Limeira. UNICAMP. 2010. p. 24.

MARCORATTI. **O processo de Software**, 2014. Disponível em: <http://www.macoratti.net/proc_sw1.htm>. Acesso em: 28 nov. 2014.

MAZZOLA, V. B. **Engenharia de Software - Conceitos Básicos**, Apostila, 2010. Disponível em: <https://jalvesnicacio.files.wordpress.com/2010/03/engenharia-de-software.pdf>. Acesso em: 17 dez. 2014.

OLIVEIRA, E. S. **Uso de Metodologias Ágeis no Desenvolvimento de Software.** Monografia. UFMG. Belo Horizonte. 2003. p. 36.

PRESSMAN, R. S. **Engenharia de Software**, 7ª edição. São Paulo: McGraw-Hill, 2011. p. 780.

PRIKLADNICKI, R.; WILLI, R.; MILANI, F.; **Métodos ágeis para desenvolvimento de software**. Porto Alegre: Bookman, 2014. p. 311.

SILVA, B. C. C. **Integrando Agilidade com maturidade**. Revista Engenharia de Software Magazine. Ed. 59. Disponível em: <http://www.devmedia.com.br/integrando-agilidade-com-maturidade-revista-engenharia-de-software-magazine-59/28197>. Acesso em: 16 dez. 2014.

SHORE, J.; WARDEN, S.; **The Art of Agile Development**. O' Reilly Media. 2008. 411p.

SOARES, M. S. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software**. Unipac – Universidade Presidente Antônio Carlos, Faculdade de Tecnologia e Ciências de Conselheiro Lafaiete. Disponível em: <http://www.dcc.ufla.br/infocomp/artigos/v3.2/art02.pdf>. Acesso em: 15 dez. 2014.

SOMMERVILLE, I. **Engenharia de Software**, 6. ed.. Editora Pearson do Brasil, 2003. p. 292.

SOMMERVILLE, I. **Engenharia de Software**, 7. ed. Editora Pearson do Brasil, 2007.

WAZLAWICK, R. S. **Engenharia de Software: Conceitos e Práticas**, 1. ed.., Elsevier, 2013. 506p.

WIKIPÉDIA. **Desenvolvimento iterativo e incremental.** Disponível em: <http://pt.wikipedia.org/wiki/Desenvolvimento_iterativo_e_incremental>. Acesso em: 17 jan. 2015.

5

Processo Unificado (UP)

Neste capítulo, veremos a abordagem de desenvolvimento de *software* que visa a construção de sistemas orientados a objetos, por meio do Processo Unificado, de forma iterativa e adaptativa.

De acordo com Canêz, (2014, o Processo Unificado tem como característica a repetição de ciclos durante o desenvolvimento de um sistema, por isso esse processo é dito como evolucionário e é subdividido em quatro fases: Concepção, Elaboração, Construção e Transição. Além destas fases, há também a subdivisão em iterações por meio de fluxos de trabalhos ou *workflows*.

Os profissionais da área de TI certamente buscam formas de produtividade para sua equipe, organização e até mesmo pessoal. Sendo assim, neste capítulo tem dois assuntos principais que certamente dão um ganho de produtividade muito grande ao profissional de TI, equipe e organização. Tratam-se das Ferramentas CASE e o estudo de um modelo de processo de *software* chamado RUP, atualmente suportado pela IBM. A IBM adquiriu várias empresas de *software*, inclusive a Rational, que originalmente concebeu o RUP. Além do RUP, conheceremos outro processo de desenvolvimento, baseado no Processo Unificado, que é o PRAXIS. Fique atento a este capítulo, pois será de grande ajuda para os seus estudos e sua vida como profissional de TI.

OBJETIVOS

- conhecer o processo unificado e suas fases;
- aprender sobre o ciclo de vida do processo unificado;
- conhecer o processo RUP;
- conhecer o processo de desenvolvimento PRAXIS;
- conhecer o que é uma ferramenta CASE;
- obter o melhor desempenho de sua equipe de desenvolvimento através de utilização e integração entre ferramentas CASE.

5.1 Processo unificado (UP)

O UP (*Unified Process* - Processo Unificado) é o primeiro modelo de processo adaptado ao uso com a UML (Unified Modeling Language), criado por pioneiros da orientação a objetos nos anos 1990. Sua criação foi baseada nas práticas de maior retomo de investimento (ROI) do mercado (WAZLAWICK, 2013, p. 119). É uma estrutura de desenvolvimento genérica que pode ser utilizada para criação de outros processos. Algumas etapas podem ser reorganizadas ou descartadas para a adequação ao volume e às características de cada sistema (MACHADO e PEREIRA, 2006, p. 3).

As principais características do Processo Unificado são:

a) processo iterativo e incremental, em que o software é incrementado de melhorias a cada iteração do ciclo de desenvolvimento;

b) orientado a caso de uso, sendo este um artefato que descreve cada funcionalidade do sistema e subsidia todas as etapas do desenvolvimento;

c) centrado na arquitetura, de forma que ao tratar a arquitetura do software seja evitado o retrabalho e aumente a capacidade de reuso de seus componentes (MACHADO e PEREIRA, 2006, p. 3);

d) focado em riscos, em função das priorizações dos casos de uso mais críticos nos primeiros ciclos iterativos" (WAZLAWICK, 2013, p. 123).

5.2 Fases do processo

O processo unificado busca aproveitar os melhores recursos e características dos modelos tradicionais de processo de *software*; e implementa diversos princípios do desenvolvimento ágil de *software* (PRESSMAN, 2011, p. 71).

No livro que deu origem ao processo unificado, Ivar Jacobson, Grady Booch e James Rumbaugh discutem a necessidade de um processo de *software* "dirigido a casos de uso, centrado na arquitetura, iterativo e incrementai" ao afirmarem:

Hoje em dia, a tendência do *software* é no sentido de sistemas maiores e mais complexos. Isso se deve, em parte, ao fato de que os computadores tornam-se mais potentes a cada ano, levando os usuários a ter uma expectativa maior em relação a eles. Essa tendência também foi influenciada pelo uso crescente da Internet para troca de todos os tipos de informação.

> Nosso apetite por *software* cada vez mais sofisticado aumenta à medida que tomamos conhecimento de uma versão do produto para a seguinte, como o produto pode ser aperfeiçoado. Queremos *software* que seja mais e mais adaptado a nossas necessidades, mas isso, por sua vez, simplesmente torna o *software* mais complexo. Em suma, queremos cada vez mais.
>
> PRESSMAN, 2011, p. 71

No processo unificado a comunicação com o cliente e os métodos racionalizados (sequencializados) para descrever a visão do cliente sobre um sistema (casos de uso) são considerados essenciais. Há ênfase na arquitetura de *software*, auxiliando na manutenção do foco nas metas corretas, tais como compreensibilidade, confiança em mudanças e reutilização. O fluxo de processo deve ser iterativo e incremental, «proporcionando a sensação evolucionária que é essencial no desenvolvimento de *software* moderno" (PRESSMAN, 2011, p. 72).

As atividades do UP são bem definidas no seguinte sentido (WAZLAWICK, 2013, p. 119):

a) Descrição clara e precisa.
b) Determinam responsáveis
c) Determinam artefatos de entrada e saída.
d) Determinam dependências entre as atividades.
e) Seguem um modelo de ciclo de vida bem definido.
f) Há uma descrição sistemática de como podem ser executadas com as ferramentas disponibilizadas (procedimentos).
g) Preconizam o uso da linguagem UML.

> "As atividades incluem *workflows*, que são grafos que descrevem as dependências entre diferentes atividades. *Workflows* estão associados às disciplinas do processo unificado, que variam de implementação para implementação"
>
> WAZLAWICK, 2013, p. 119.

As **fases do processo unificado** podem ser divididas em:

a) **Concepção (inception)**: trata-se da elaboração de uma visão abrangente do sistema. Envolve a comunicação com o cliente e o planejamento. Há a identificação das necessidades de negócio para o software; propondo uma

arquitetura rudimentar para o sistema e desenvolvendo um planejamento para a natureza iterativa e incremental do projeto. Nessa fase são levantados os principais requisitos do sistema e, é elaborado um modelo conceitual preliminar e são identificados os casos de uso. Assim, é medido o esforço de desenvolvimento dos casos de uso e construído o plano de desenvolvimento, composto por um conjunto de ciclos iterativos nos quais são acomodados os casos de uso. Posteriormente, a arquitetura será refinada e expandida para um conjunto de modelos que representam visões diferentes do sistema. (PRESSMAN, 2011, p. 72; WAZLAWICK, 2013, p. 123). "O planejamento identifica recursos, avalia os principais riscos, define um cronograma e estabelece uma base para as fases que serão aplicadas à medida que o incremento de software é desenvolvido. Pode haver alguma implementação e teste, caso seja necessário elaborar protótipos para redução de risco" (PRESSMAN, 2011, p. 72).

b) **Elaboração (elaboration)**: nessa fase, as iterações têm como objetivo detalhar a análise, expandindo os casos de uso, promovendo o detalhamento e antecipando situações excepcionais, com fluxos alternativos. O modelo conceitual preliminar torna-se um modelo definitivo, sobre o qual serão aplicados padrões de análise e uma descrição funcional poderá ser feita, bem como o *design* lógico e físico do sistema.

c) **Construção (construction)**: nesta fase, os casos de uso mais complexos já foram tratados e a arquitetura já foi estabilizada. Assim, as atividades consistem prioritariamente na geração de código e testes do sistema. A geração de código de forma automatizada e a introdução de modelos de desenvolvimento dirigidos a teste devem proporcionar um código de alta qualidade.

d) **Transição (deployment)**: a fase de transição se traduz no processo de implementar o sistema no ambiente final, com a realização de testes de operação. É realizada a transferência de dados de possíveis sistemas antigos para o novo sistema e também ocorre o treinamento de usuários. Nessa fase pode haver ainda alguma adaptação, revisão de requisitos e geração de código, mas não de forma significativa.

> Apesar de as fases do Processo Unificado terem diferentes ênfases, espera-se que cada ciclo iterativo tome um conjunto de casos de uso e os desenvolva desde os requisitos até a implementação e a integração de código final.
>
> WAZLAWICK, 2013, p. 124.

Na fase de elaboração, a equipe dedicará mais tempo a questões de análise e projeto do que de implementação e teste. Na fase de construção, grande parte dos requisitos já terão sido desvendados e o esforço se concentrará nas atividades de programação. A cada fase do UP, um macro-objetivo (milestone) é alcançado.

Na fase de **concepção,** o objetivo é compreender o escopo do projeto, bem como planejar seu desenvolvimento. Na fase de **elaboração,** os requisitos devem ter sido entendidos e uma arquitetura estável deve ter sido definida. Na etapa de **construção,** o sistema deve ter sido programado e testado. Ao final da fase de **transição,** o *software* deve estar instalado e deve estar sendo utilizado pelos usuários finais (WEST, 2002; WAZLAWICK, 2013, p. 124).

CONEXÃO

Seguem alguns *links* de ferramentas que são utilizadas no processo unificado e que estão relacionadas com os assuntos tratados neste capítulo:

- System Architect: http://migre.me/oHSyr
- Rational Rose: http://migre.me/oHSxQ
- Oracle Designer: http://migre.me/oHSx3
- GDPro:

5.3 Ciclo de vida do processo

A figura 5.1 mostra o processo unificado, que envolve a repetição de uma série de ciclos ao longo da vida de um sistema.

Cada ciclo é concluído com uma versão do produto pronta. “Essa versão é um conjunto relativamente completo e consistente de artefatos, possivelmente incluindo manuais e um módulo executável do sistema, que podem ser distribuídos para usuários internos ou externos” (MARTINS, 2014).
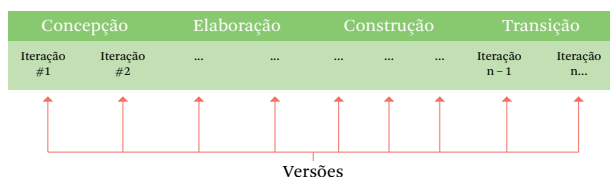
Figura 5.1 – Ciclos do Processo Unificado. Fonte: MARTINS, 2014.

Cada ciclo consiste de quatro fases: início, elaboração, construção e transição. Cada fase é também subdividida em iterações, como discutido anteriormente (figura 5.1).

Estas 4 fases, subdivididas em iterações, passam por cinco fluxos de trabalho (*Workflow*): Requisitos, Análise, Projeto, Implementação e Teste. A figura a seguir mostra um gráfico deste fluxo, também conhecido como gráfico das baleias (CANÊZ, 2014).
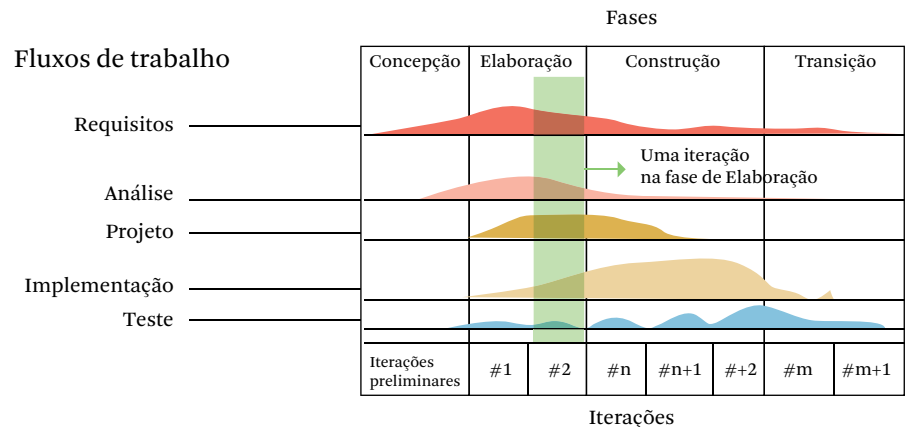
Figura 5.2 – Fases, Fluxo de trabalho (*workflows*) e Iterações no Processo Unificado. Fonte: CANÊZ, 2014.

Há um incremento do projeto a cada iteração, valendo-se das informações dos usuários que já estão utilizando os sistemas, assim como as informações obtidas nas interações anteriores. "No processo unificado cada iteração pode ser considerada um projeto de duração fixa, sendo que cada um destes inclui suas próprias atividades de análise de requisitos, projeto, implementação e testes" (CANÊZ, 2014).

O resultado de cada iteração é um sistema executável, mesmo não estando totalmente completo tem qualidade de produto final e não de protótipo (CANÊZ, 2014).

5.3.1 Fluxo de requisitos

A análise de requisitos é o primeiro passo de uma iteração (CANÊZ, 2014).

> Os requisitos do sistema são especificados no processo de identificação das necessidades de usuários e clientes. Os requisitos são representados como casos de uso por meio do modelo de casos de uso. "Os casos de uso são representados através da notação UML, onde cada caso de uso é composto pelos diagramas de casos de uso que compõem o sistema" (CANÊZ, 2014).

Os requisitos mais importantes são identificados durante a fase de concepção. Na fase de elaboração, os requisitos remanescentes são analisados, possibilitando a compreensão real do tamanho do sistema. Ao final da fase de elaboração, a maior parte dos requisitos do sistema devem ter sido descritos, no entanto apenas 10% destes requisitos são implementados nesta fase. Eles serão identificados e implementados durante a fase de construção. Já na fase de transição quase não há requisitos a serem identificados, a menos que ocorram alterações nos requisitos (Canêz, 2014).

O fluxo (ou *workflow*) de requisitos tem como objetivo planejar o desenvolvimento em direção ao sistema correto. Busca, assim:

I. Entender o contexto do sistema

II. Para expressar o contexto de um sistema, o gerente de projeto define: um modelo do domínio; um modelo do negócio (casos de uso e objetos do negócio).

III. Enumerar os requisitos candidatos

IV. Ideias mencionadas pelos clientes, usuários, analistas e desenvolvedores são enumeradas como requisitos candidatos.

V. Capturar os requisitos funcionais (através de casos de uso)

  1. Identificar os atores e casos de uso (diagramas de casos de uso)
  2. Dar prioridades aos casos de uso
  3. Detalhar os casos de uso
  4. Definir a interface com o usuário
     a) projeto da interface lógica: elementos que representam os atributos dos casos de uso.
     b) projeto e protótipo da interface física: *sketches* com os elementos identificados na interface lógica. Protótipos executáveis são construídos para as partes mais importantes e onde a usabilidade deve ser avaliada.

VI. Capturar os requisitos não funcionais

5.3.2 Fluxo de análise

A análise é o segundo elemento do fluxo de trabalho de uma iteração.

O produto gerado no fluxo de análise é o modelo de análise, que refina os requisitos especificados no fluxo de requisitos por meio da criação de diagramas de classes conceituais. Assim, é possível identificar o funcionamento interno do sistema. O diagrama de iterações e o diagrama de gráficos de estados que representam a dinâmica do sistema são gerados no modelo de análise. Assim, torna-se mais fácil a definição de uma arquitetura estável, facilitando também o entendimento detalhado dos requisitos. No modelo de análise é dado o primeiro passo para o desenvolvimento do modelo de projeto (Canêz, 2014).

Assim, pode-se afirmar que no fluxo de análise é gerado o modelo de análise que descreve as características comportamentos e estruturais do sistema em um nível conceitual

O fluxo de análise tem maior importância durante a fase de elaboração.

> Para realizar esse fluxo de trabalho corretamente é necessário primeiro identificar e detalhar os casos de uso para uma iteração, e depois, através da análise da descrição de cada caso de uso, sugerir quais classes e relacionamentos são necessários para realiza-lo (CANÊZ, 2014).

O fluxo (ou *workflow*) de análise: tem como objetivo refinar e estruturar os requisitos levantados no *workflow* de requisitos. Busca, assim:

I. Analisar a arquitetura (arquiteto)
   1. Identificar os pacotes de análise
   2. Identificar as classes de entidade óbvias
   3. Identificar os requisitos especiais comuns que surgem durante a análise (ex.: distribuição e concorrência, características de segurança)

II. Analisar um caso de uso (engenheiro de caso de uso)
   1. Identificar as classes de análise do caso de uso
   2. Descrever as interações entre os objetos de análise

III. Analisar uma classe (engenheiro de componente)
   1. Identificar as responsabilidades através dos papéis que ela desempenha nas realizações de todos os casos de uso nos quais ela participa.

a) Exemplo: Responsabilidades da classe Agendador de pagamento:

2. Criar uma requisição de pagamento.

3. Monitorar os pagamentos que foram agendados e enviar uma notificação quando o pagamento foi efetivado ou cancelado.

IV. Identificar os atributos, associações e agregações, generalizações

V. Capturar os requisitos especiais

VI. Analisar um pacote (engenheiro de componente)

1. Inclui a descrição das dependências.

Diferentes maneiras de aplicar a análise:

1. O modelo de análise é utilizado para descrever os resultados da análise e é mantida a consistência dele durante todo o ciclo de vida do *software*.

2. O modelo de análise é utilizado para descrever os resultados da análise, mas ele é visto como uma ferramenta temporária e intermediária. Durante as fases de projeto e implementação, o modelo de análise não é mais mantido.

3. O modelo de análise não é utilizado para descrever os resultados da análise.

5.3.3 Fluxo de projeto

O projeto é o terceiro elemento do fluxo de trabalho de uma iteração. O modelo de projeto é construído com base no modelo de análise definido no fluxo de análise (CANÊZ, 2014).

No fluxo de projeto o sistema é moldado e sua forma é definida, atendendo às necessidades especificadas pelos requisitos. O modelo de projeto é desenvolvido, descrevendo o sistema em um nível físico. O principal objetivo deste fluxo é obter a compreensão detalhada dos requisitos do sistema, considerando fatores como linguagens de programação, so, tecnologias de banco de dados, interface com o usuário (Canêz, 2014). “O trabalho realizado no fluxo de projeto é mais concentrado entre o fim da fase de elaboração e o início da fase de construção, como pode ser observado na figura anterior” (Canêz, 2014).

O fluxo (ou *workflow*) de projeto: tem como objetivo criar um plano detalhado (*blueprint*) para o modelo de implementação. Busca, assim:

I. Projetar a arquitetura (arquiteto)
   1. Identificar os nodos e suas configurações de rede
   2. Identificar os subsistemas e suas interfaces
   3. Identificar as classes de projeto significantes (ex.: identificação das classes de projeto a partir das classes de análise)
II. Projetar um caso de uso (engenheiro de caso de uso)
   1. Identificar as classes de projeto do caso de uso (diagrama de classes)
   2. Descrever as interações entre os objetos (diagramas de sequência)
   3. Identificar os subsistemas participantes que contêm classes de projeto que participam do caso de uso em questão
   4. Capturar os requisitos de implementação (ex.: um objeto deverá ser capaz de manipular 10 clientes compradores sem um *delay* perceptível.)
III. Projetar uma classe (engenheiro de componente)
   1. Identificar as operações, os atributos, as associações e agregações e as generalizações.
   2. Descrever os métodos: podem ser especificados usando linguagem natural ou pseudocódigo, mas na maioria das vezes, métodos não são especificados durante o projeto.
   3. Descrever os estados: para os objetos de projeto que são controlados pelo estado, é importante usar um diagrama de *statechart*.
IV. Projetar um subsistema (engenheiro de componente)
   1. Assegurar que o subsistema satisfaz a realização das operações definidas pelas interfaces que ele provê.

5.3.4 Fluxo de implementação

O fluxo de implementação é baseado no modelo de projeto; e tem como objetivo implementar o sistema em termos de componentes, ou seja: código-fonte , arquivos executáveis etc. A arquitetura do sistema é definida prioritariamente durante o fluxo de projeto, que tem como produto final um modelo de implementação.

O modelo de implementação busca: planejar as integrações do sistema em cada iteração, resultando em um sistema implementado como um sucessão de etapas pequenas e gerenciáveis; implementar os subsistemas encontrados durante o fluxo de projeto; testar e integrar as implementações, compilando-as em arquivos executáveis, antes de enviá-las ao fluxo de teste (CANÊZ, 2014).

O fluxo de implementação tem maior importância durante a fase de construção. Há maior simplicidade neste fluxo, pois as decisões mais difíceis já foram tomadas durante a etapa de fluxo de projeto. O código gerado durante a implementação deve ser uma simples tradução das decisões de projeto em uma linguagem especifica (CANÊZ, 2014).

O fluxo (ou *workflow*) de Implementação: tem como objetivo implementar o sistema a partir do resultado do projeto. Busca, assim:

I. Arquitetar a implementação (arquiteto)
   1. As classes de projeto mais significantes para a arquitetura são identificadas e mapeadas para os nodos.

II. Integrar o sistema
   1. Criação de um plano de integração que descreve os *builds* de uma iteração e como serão integrados.

III. Implementar um subsistema (engenheiro de componente)

IV. Implementar uma classe (engenheiro de componente)

V. Executar os testes de unidade (engenheiro de componente)

VI. Teste de especificação (teste da caixa preta) e teste de estrutura (teste da caixa branca).

VII. Também são feitos testes de performance e capacidade.

5.3.5 Fluxo de teste

O fluxo de teste é desenvolvido com base no produto gerado durante o fluxo de implementação. "Os componentes executáveis são testados para só então serem disponibilizados ao usuário final. Os componentes testados que apresentarem problema retornarão a fluxos anteriores, onde serão corrigidos" (CANÊZ, 2014).

O teste de um sistema é realizado durante a fase de elaboração, quando a arquitetura do sistema é definida, e durante a fase de construção, quando o sistema é implementado. Um planejamento dos testes deve ser feito na fase de concepção. O fluxo de testes na fase de transição se traduz em consertar defeitos encontrados durante a utilização inicial do sistema (CANÊZ, 2014).

"Durante o fluxo de teste é gerado o modelo de teste, esse modelo descreve como componentes executáveis, provenientes do fluxo de implementação, serão testados. No modelo de testes pode vir descrito com os aspectos específicos do sistema serão testados, como, por exemplo, se a interface como o usuário é simples e consistente ou se o manual de usuário cumpre o seu objetivo. Resumindo, o papel do fluxo de teste é verificar se os resultados do fluxo de implementação cumprem os requisitos estipulados por clientes e usuários, para decidir se o sistema necessita de revisões ou se o processo de desenvolvimento pode continuar" (CANÊZ, 2014).

O fluxo (ou *workflow*) de teste: tem como objetivo testar o resultado da implementação. Busca, assim:

I. Planejar e projetar o teste (projetista de teste)

    1. Planejamento dos testes de uma iteração: como e quando rodar, quando terminar os testes, estimar os recursos humanos e de sistema necessários, agendar os testes.

II. Os casos de testes são baseados nos diagramas de iteração dos casos de uso.

III. Implementar o teste (engenheiro de componente)

IV. Executar o teste de integração (testador de integração)

V. Executar o teste de sistema (testador de sistema)

VI. Avaliar o teste (projetista de teste)

5.4 RUP (Rational Unified Process)

RUP (*Rational Unified Process* ou Processo Unificado da Rational) é um processo de Engenharia de *software* criado pela Rational *Software* Corporation, adquirida pela IBM, ganhando um novo nome: IRUP (IBM Rational Unified Process).

Trata-se de uma metodologia completa de desenvolvimento de *software* que fornece várias técnicas a serem seguidas pelos gerentes e membros da equipe de desenvolvimento de *software* com o objetivo de aumentar a sua produtividade.

Como é muito apoiado em ferramentas CASE, vamos passar pelos principais tópicos existentes.

O RUP usa a orientação a objetos em sua concepção e é projetado e documentado com a UML (*Unified Modeling Language*) para ilustrar os processos em ação.

Atualmente, o RUP é controlado e mantido pela IBM, porém sua origem se deve à empresa Rational, que conseguiu muito destaque no mercado mundial por ter uma metodologia bastante consistente aliada a uma ferramenta que a suportava inteiramente (o Rational Rose). Sendo assim, e por ser adotado em várias empresas no mundo todo, usa técnicas e práticas aprovadas comercialmente.

Segundo o *site* da IBM (IBM, 2012) e Sommerville (2007), o RUP utiliza seis práticas como linha de base nos ciclo de produção. São elas:

GESTÃO DE REQUISITOS	deve-se documentar explicitamente os requisitos do cliente e acompanhar as mudanças.
ARQUITETURA BASEADA EM COMPONENTES	estruturar a arquitetura do sistema baseada em componentes, como foi visto nesta unidade
DESENVOLVIMENTO INTERATIVO	planejar os incrementos de *software* com base nas prioridades do cliente, desenvolver e entregar em primeiro lugar as características do sistema de maior prioridade no processo de desenvolvimento
MODELAGEM VISUAL DOS COMPONENTES	usar a UML principalmente para apresentar as visões estática e dinâmica do *software*
VERIFICAÇÃO DE QUALIDADE DO *SOFTWARE*	garantir que o *software* atenda às especificações inicialmente previstas
GESTÃO E CONTROLE DE MUDANÇAS	gerenciar as mudanças do *software* por meio de um sistema de gerenciamento de mudanças, procedimentos e ferramentas de gerenciamento de configuração.

O RUP particiona o ciclo de vida do processo em quatro etapas, que indicam a ênfase dada ao projeto em cada momento.

CONCEPÇÃO (INCEPTION)	contém os *workflows* necessários que as partes interessadas (*stakeholders*) concordem com os objetivos, arquitetura e o planejamento do projeto. Se as partes interessadas tiverem bons conhecimentos, pouca análise será requerida então. Se não tiverem o conhecimento necessário, mais análise será requerida.
ELABORAÇÃO (ELABORATION)	é apenas para o projeto do sistema, buscando complementar o levantamento/documentação dos casos de uso; voltado para a arquitetura do sistema, revisa a modelagem do negócio para os projetos e inicia a versão do manual do usuário. Deve-se aceitar – Visão geral do produto (incremento + integração) está estável? O plano do projeto é confiável? Custos são admissíveis?
CONSTRUÇÃO (CONSTRUCTION)	começa o desenvolvimento físico do *software*, produção de códigos, testes alfa e beta. Deve-se aceitar testes e processos de testes estáveis e se os códigos do sistema constituem “baseline”.
TRANSIÇÃO (TRANSITION)	ocorre a entrega (“*deployment*”) do *software*, é realizado o plano de implantação e entrega, acompanhamento e qualidade do *software*. Produtos (releases, versões) devem ser entregues e ocorrer a satisfação do cliente.

A figura 5.3 apresenta a estrutura do RUP; apesar de parecer um modelo em cascata, na verdade cada fase é composta de uma ou mais interações, o que se assemelha a um modelo em espiral. Estas iterações são em geral curtas (1-2 semanas) e abordam algumas poucas funções do sistema. Isto reduz o impacto de mudanças, pois quanto menor o tempo, menor a probabilidade de haver uma mudança neste período para as funções em questão.
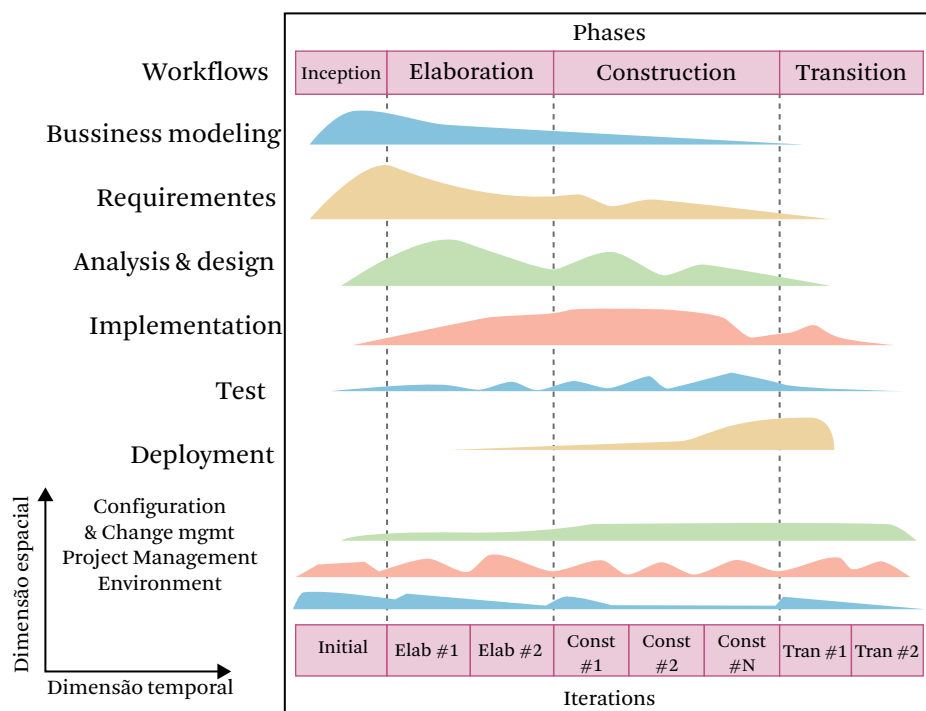
Figura 5.3 – O Processo RUP.

CONEXÃO

A seguir, apresentamos alguns *links* a respeito do processo RUP:

- http://www.wthreex.com/rup/portugues/index.htm. Excelente material on-line sobre o processo RUP, com todos os seus conceitos e características
- http://www-01.ibm.com/*software*/awdtools/rup/. Em inglês. *Site* da IBM contendo vários *links* sobre RUP e sua integração com as ferramentas de *software* da IBM. Vale a pena dar uma olhada.
- http://javafree.uol.com.br/artigo/871455/;. Este artigo relaciona qualidade de *software*, vista nas unidades anteriores, com o processo RUP.

5.4.1 Conteúdo do RUP

O RUP, além de uma metodologia, como vimos, faz parte de um *framework* oferecido pela IBM e possui uma grande documentação disponível na Internet (IBM, 2012). Desta documentação, os tópicos abaixo merecem destaque:

WORKFLOWS	cada *workflow* é detalhadamente descrito e apresenta passo a passo as tarefas, subprodutos a serem gerados e papéis de profissionais envolvidos. Cada tarefa, subproduto e papel é descrito detalhadamente. Este modelo pode ser seguido à risca ou adaptado conforme a necessidade específica.
TAREFAS	cada tarefa é descrita detalhadamente, incluindo qual papel é responsável por ela, a qual *workflow* ela pertence e quais são os subprodutos de entrada e saída.
MODELO DE EQUIPE	os diversos papéis necessários no projeto são também descritos detalhadamente. Cada papel pode ser representado por mais de um ator ou um ator pode ter mais de um papel, dependendo da carga de trabalho necessário. Ao todo são mais de 30 tipos de papéis.
MODELOS DE DOCUMENTOS	o RUP apresenta modelos e exemplos para os diversos documentos (artefatos) gerados no decorrer do projeto. Estes documentos são descritos detalhadamente assim como as tarefas que os geram e as que os utilizam. Os documentos são totalmente compatíveis com a UML, reforçando a questão de padronização.

5.4.2 Como adotar

Com base no que foi visto, a implantação do RUP pode ser feita de várias maneiras.

De uma forma bem rigorosa, seria usar o RUP à risca, ou seja, aplicar todos os métodos e processos exatamente como são propostos. A vantagem desta abordagem é que como nada será alterado, o RUP fornece todo o suporte documentalmente. Porém, existe um preço a ser pago, pois o RUP na íntegra é complexo, sendo assim é uma implantação que corre um certo risco. Esta abordagem implicaria em treinamentos, projetos piloto etc. Existem propostas de projetos de adoção do RUP descritas no próprio produto.

De outra forma, seria adotar outro modelo de processo mais simples ou conhecido (o atual, se existir) e ter o material do RUP como fonte de referência complementar para assuntos não abordados no modelo oficialmente adotados como, por exemplo, os modelos de documentos.

A primeira abordagem é interessante para empresas que precisam de uma grande formalização e padronização do processo de desenvolvimento de *software* e cujo método atual seja totalmente inadequado ou inexistente.

A segunda forma seria interessante para quem já tem alguma metodologia que considera adequada e segura mas possui deficiência em alguma área, como, por exemplo, o suporte à UML.

Existem soluções intermediárias que também são possíveis.

Segundo Sommerville (2007), o RUP não é um processo adequado a todos os tipos de desenvolvimento, porém representa uma nova geração de processos genéricos. A maior inovação é a separação de fases e *workflows*, e o reconhecimento de que a implantação de *software* no ambiente do usuário é parte do processo. As fases são dinâmicas e objetivas. Os *workflows*, são estáticos e constituem atividades técnicas que não estão associadas a uma fase, mas podem ser usadas ao longo do desenvolvimento para alcançar os objetivos de cada fase.

5.4.3 PRAXIS

O Praxis, apresentado por Paula Filho (2001), é um processo de desenvolvimento de *software* baseado no processo unificado. Sua simplificação serve de suporte a treinamentos de engenheiros de *software*.

O Práxis foi projetado para faciliar o ensino de processos de *software*. Tem, assim, enfoque educacional, busca dar suporte ao treinamento em Engenharia

de *Software* e à implantação de processos em organizações de desenvolvimento de *software*.

> "O objetivo educacional é exposição das técnicas mais relevantes, além do treinamento eficaz e eficiente. É importante reiterar que as técnicas em estágio de pesquisa, técnicas usadas apenas por organizações de alta tecnologia, em vias de obsolescência não são incluídas" (INFOXZONE, 2010).

É adequado para projetos menores com prazo de realização de um ano. "Seu autor sugere sua customização a partir da observação das especificidades de cada projeto, podendo ser adicionados ou removidos alguns de seus elementos" (MACHADO e PEREIRA, 2006, p. 3).

Compreende várias etapas e diversos artefatos em cada etapa. Artefatos são o produto de uma ou mais atividades no desenvolvimento de um *software* ou sistema.

O Praxis propõe três tipos de artefatos: modelos, documentos e relatórios (MACHADO e PEREIRA, 2006, p. 3).

A cada iteração no RUP haverá um documento que servirá de fonte de informação e / ou discussão, promovendo o surgimento de novo documento ou nova versão do documento fonte utilizado (INFOXZONE, 2010).

> "As referências de processo se baseiam no RUP (IBM Rational Process Unified), e dentre suas características, podemos citar: mesmas raízes do processo unificado, estrutura diferente de disciplinas, coleção de processos concretos, referência industrial de fato. Também se baseiam em XP (Extreme Programming), que é mais conhecido de processos ágeis, desenvolvimento dirigido por testes, planejamento baseado em liberações pequenas" (INFOXZONE, 2010).

O Praxis propõe um ciclo de vida do *software* em quatro fases:

a) **Concepção**: em uma única iteração, é realizado o levantamento de requisitos a fim de verificar a viabilidade do projeto;

b) **Elaboração**: nesta fase são elaborados os modelos do sistema, que devem expressar seus requisitos, funcionalidades e interfaces, bem como os primeiros esboços do desenho do software;

c) **Construção**: nesta etapa deve ser encerrado o desenho do sistema e, a partir dele, a construção efetiva do software e seus primeiros testes.

d) **Transição**: fase em que deverão ser realizados os últimos testes e liberada a primeira versão do software para o cliente. (MACHADO e PEREIRA, 2006, p. 3)

5.5 Ferramentas CASE

Segundo Pressman (2006), CASE é a abreviação para *Computer Aided Software Engineering* – Engenharia de *Software* Auxiliada por Computador.

Porém, esta palavra (CASE) tem sido interpretada de diversas maneiras no mercado de *software*, pois estes oferecem *software* erroneamente classificados como ferramentas CASE de fato. Numa primeira análise, o termo CASE não fala exatamente em ferramentas, mas se olharmos a expressão *auxiliado por computador*, tal só pode ser conseguido pela utilização de ferramentas informáticas.

Sommerville (2007) define CASE como um:

> conjunto de técnicas e ferramentas de informática que auxiliam o engenheiro de *software* no desenvolvimento de aplicações, com o objetivo de diminuir o respectivo esforço e complexidade, de melhorar o controle do projeto, de aplicar sistematicamente um processo uniformizado e de automatizar algumas atividades, a verificação da consistência e qualidade do produto final e a geração de artefatos.

Os artefatos podem ser: documentação, diagramas, modelos de dados, e até mesmo o código-fonte . Uma ferramenta CASE não é mais do que um produto destinado a suportar uma ou mais atividades de engenharia de *software* relacionadas com uma ou mais metodologias de desenvolvimento. Atualmente encontramos *softwares* que atendem todo o processo de desenvolvimento, desde a fase de requisitos até o acompanhamento de sua mantenção.

5.5.1 Histórico

Apenas no início da década de 1980 é que surgiram no mercado as primeiras ferramentas que se consideram atualmente como CASE. O Excelerator, uma das primeiras ferramentas CASE unanimemente considerada como tal, surgiu

em 1984. A crescente importância que foram tendo no processo de desenvolvimento está diretamente relacionada com um conjunto de fatores decisivos que contribuíram para o crescente sentimento da necessidade deste tipo de ferramentas. Veja a seguir:

- A mudança de ênfase das atividades de programação para atividades de análise e desenho de *software*, de modo a superar os diversos problemas dos métodos de trabalho ad-hoc.
- Utilização de computadores pessoais e de interfaces de trabalhos gráficos.
- O aparecimento de diversas técnicas de modelagem de sistemas, que implicavam no desenho de diagramas gráficos (tais como os fluxogramas ou diagramas de fluxos de dados), em que a representação destas notações em papel, ou em textuais, tornava-se impraticável à medida que a respectiva complexidade aumentava.
- O aumento da complexidade e do tamanho do *software*, associado às maiores capacidades do *hardware*.

No início dos anos 1990, muitas das ferramentas CASE passaram a ser designadas por ferramentas RAD (Rapid Application Development), o que traduzia a preocupação de aumentar o ritmo do desenvolvimento de aplicações. Exemplos destas ferramentas foram (e algumas ainda são) o Visual Basic, Sql Windows, Dephi, Powerbuilder, Oracle Designer e Oracle Developer, que são frequentemente utilizadas para auxiliar o desenvolvimento de *software* para ambientes cliente-servidor. A figura 5.4 apresenta a evolução das ferramentas.
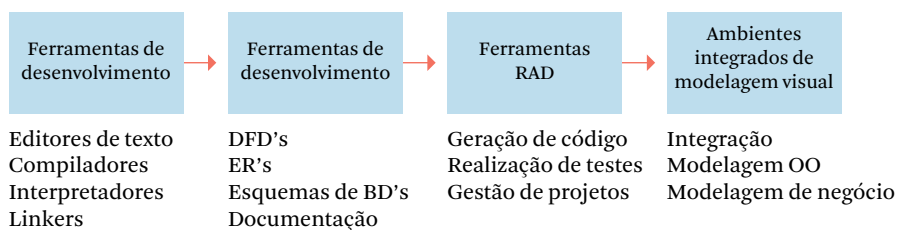
Figura 5.4 – Evolução das ferramentas CASE.

A partir de meados dos anos 1990, com a crescente importância das abordagens orientadas a objetos e o desenvolvimento de componentes, a terminologia utilizada por alguns autores passou a ser ferramentas de modelagem visual. No entanto, a expressão CASE permanece, no nosso entender, como a mais abrangente de todas.

A evolução dos conceitos e ferramentas CASE nem sempre foi um caminho simples e bem-sucedido. Por exemplo, a iniciativa da IBM designada por AD/Cycle fracassou em 1992. Esta iniciativa tinha por objetivo definir um conjunto de especificações-padrão, que permitissem a todas as ferramentas de desenvolvimento comunicar-se entre si. Igualmente, a grande maioria dos primeiros

fabricantes deste tipo de ferramentas já não existe ou foi adquirida por outras empresas.

Mais recentemente, a introdução dos conceitos da orientação a objetos veio de alguma forma revolucionar o mercado, quer porque uma parte significativa das ferramentas tradicionais teve que se "reinventar" e incorporar novas técnicas de modelagem integradas (ou não) com as abordagens estruturadas já existentes (é o caso do System Architect), quer porque surgiram no mercado novas ferramentas que suportam exclusivamente este paradigma (é o caso da ferramenta Rose da Rational). Neste contexto, assume particular destaque o UML, que vem assumindo um papel crescente ao nível das notações de modelação. Hoje em dia, praticamente todas as ferramentas que detêm uma quota de mercado mais significativa incorporam algum suporte para o UML.

Esta é uma das "áreas quentes" das ferramentas CASE que se concentram na modelagem de *software*: a dúvida sobre a preponderância de determinadas notações e consequente implementação pelas ferramentas.

5.5.2 Arquitetura de ferramentas

A maioria das ferramentas CASE especializa-se sobretudo numa tarefa específica do processo de desenvolvimento de *software*. Algumas concentram-se na disponibilização de funcionalidades relevantes para a fase de concepção (por exemplo, elaboração de diversos diagramas), enquanto outras estão particularmente direcionadas para a fase de implementação (por exemplo, desenvolvimento visual, geração de código e apoio à realização de testes).

> O termo repositório designa o componente da arquitetura das ferramentas CASE que é utilizado como meio de armazenamento, gestão e partilha de objetos, modelos, documentos, ou quaisquer outros artefatos produzidos por algum dos restantes componentes que completam a arquitetura.

A arquitetura típica das ferramentas CASE (ou pelo menos uma que se considera mais adequada para este tipo de aplicações) é constituída por um conjunto de aplicações/componentes, suportados por um repositório integrado.

Na prática, o papel do repositório pode ser concretizado através de uma base de dados, como é o caso típico dos fornecedores que possuem simultaneamente produtos CASE e bases de dados (casos da Oracle e da Sybase), mas muitos produtos utilizam um simples sistema de gestão de arquivos, alguns com formatos proprietários.

O repositório de uma ferramenta CASE é particularmente relevante, uma vez que facilita a gestão de modelos elaborados, e a respectiva reutilização, disponibilizando, para isso, mecanismos potentes de pesquisa.

Tipicamente, o seu conteúdo incluirá:

- Templates de âmbito variado, diagramas e documentos, que facilitam a elaboração de artefatos concretos a partir de modelos genéricos.
- Templates e *frameworks* de aplicação, a partir dos quais podem ser construídos *esqueletos* de aplicações em função de um conjunto de parâmetros.
- Bibliotecas de objetos, classes e componentes, que além de eventuais componentes que possam vir inicialmente com as ferramentas, permitem a integração de outros desenvolvidos ao longo do tempo.
- Diagramas diversos que resultam da modelação do sistema.
- Código-fonte , programas executáveis e aplicações empacotadas prontas para serem distribuídas aos usuários finais.
- Arquivos de dados para testes e *scripts* de execução dos mesmos.

O repositório apresenta as funcionalidades típicas de um sistema de gestão de bases de dados, no que diz respeito a:

- Garantia de integridade de dados.
- Partilha de informação.
- Suporte ao trabalho concorrente de vários utilizadores.
- Facilidades de realização de operações de pesquisa.

O repositório é um componente crítico ao oferecer mecanismos e estruturas de dados para a integração entre as ferramentas, constituindo-se como o meio de armazenamento comum, para todos os artefatos. Com objetivo de garantir o

sucesso do repositório, como facilitador do compartilhamento de informação, é necessário que sejam definidos adicionalmente os seguintes aspectos:

- Um formato comum para troca de informação descritiva dos artefatos.
- Uma interface comum para recuperar e utilizar os artefatos.

CONEXÃO

Veja, a seguir, os *links* de algumas ferramentas citadas neste capítulo:

- System Architect: http://www-01.ibm.com/*software*/awdtools/systemarchitect/
- Rational Rose: http://www-01.ibm.com/*software*/awdtools/developer/rose/
- Oracle Designer: http://www.oracle.com/technetwork/developer-tools/designer/overview/index-082236.html
- GDPro: http://www-d0.fnal.gov/computing/tools/gdpro/gdpro.html
- Power Designer: http://www.sybase.com.br/products/modelingdevelopment/powerdesigner
- Silverrun: http://www.silverrun.com/
- ERWin: http://erwin.com/
- Genexus: http://www.genexus.com/Inicio/inicio?pt

5.5.3 Integração entre ferramentas

Existem outras possibilidades para realizar a integração entre ferramentas CASE além do repositório. Uma delas e talvez a mais simples é o compartilhamento de informação entre duas ferramentas distintas por meio da exportação e importação de dados utilizando formatos universais (por exemplo, campos de texto separados por vírgulas, segundo formato tipo CSV). Esta alternativa, no entanto, não garante qualquer integridade da informação, uma vez que o mesmo conceito pode estar duplicado, assim não será possível garantir a coerência da informação: quaisquer alterações numa das cópias não são refletidas nas restantes.

XMI (XML Metadata Interchange) é um padrão definido pela OMG que representa os modelos UML utilizando-se XML. Ele é definido a partir de um documento DTD XML, de 121 páginas, que permite representar os modelos UML em forma textual.

O CASE Data Interchange Format (CDIF) é outra iniciativa desenvolvida nesta área, conduzida pela Electronic Industries Association. O CDIF começou a ser desenvolvido em 1987, tendo sido originalmente publicado em 1991 e revisto em 1994 e em 1997, de forma a incorporar contributos posteriores. O CDIF consiste numa família de padrões, que definem: (1) uma arquitetura única para a troca de informação entre ferramentas de modelagem de diferentes fabricantes e (2) as interfaces entre os componentes que implementam esta arquitetura.

Durante a década de 1990, muitas ferramentas aderiram a este padrão, mas com a crescente utilização das notações orientadas a objetos e, em particular, com o UML, perdeu sua importância e sucesso que lhe eram atribuídos inicialmente. Para isso, muito contribuiu para o desenvolvimento do XMI, como novo padrão de integração entre ferramentas que suportam UML, e baseado em XML.

5.5.4 Taxonomia

A inclusão de um produto no universo das ferramentas CASE depende da abrangência da definição considerada. Por exemplo, uma parte significativa das pessoas teria dificuldade em considerar um *simples* editor de texto como uma ferramenta CASE. Normalmente, pensamos sempre em ferramentas de suporte a atividades mais "nobres", e a edição de código não traz aparentemente qualquer valor acrescentado ao desenvolvimento de *software*.

No entanto, se pensarmos no significado exato do termo, e que sem um editor de texto não seria possível produzir um programa (para além de que ele pode mesmo automatizar algumas tarefas, como sejam a substituição de palavras), seremos levados a reconsiderar e a incluí-lo nesta classificação.

Os critérios utilizados para classificar as ferramentas CASE são muito diversos. Os mais significativos incluem:

- a análise das funcionalidades disponíveis;
- o papel que representam para os gestores ou para elementos técnicos;
- a possibilidade de serem utilizados nas várias fases do processo de desenvolvimento de software.

Uma primeira classificação das ferramentas CASE pode ser efetuada com base nos seguintes critérios:

- Fases do processo de desenvolvimento nas quais as ferramentas se aplicam:
    - **Ferramentas** – *Upper-Case* são aplicações que se especializaram na fase de concepção do *software* (ferramentas de análise e especificação e/ou modelação de requisitos).
    - **Ferramentas** – *Lower-Case* são aplicações utilizadas na fase de implementação (ferramentas de desenho técnico, de edição e compilação de código e de testes).
- Utilização das ferramentas em atividades específicas de uma fase/tarefa ou concebidas para atividades que se desenrolam ao longo de todo o ciclo: um exemplo típico das primeiras são as ferramentas que implementam ambientes de desenvolvimento, enquanto que nas segundas se pode incluir uma ferramenta de gestão de projetos.

Uma outra classificação mais detalhada agrupa as ferramentas nas seguintes categorias:

- **Modelagem de processos de negócio** – ferramentas orientadas para a análise e especificação dos processos de negócio, que permitem verificar como os objetivos estratégicos de negócio são concretizados nos processos. Além de utilizarem diversas notações e diagramas para a representação de informação do negócio (cadeia de valor, responsabilidades e funções da organização), recorrem com frequência a técnicas de simulação e análise de custos. Estas funcionalidades possibilitam a utilização destas ferramentas para compreender o funcionamento da empresa e identificar problemas ou oportunidades de melhoria (e não para o desenvolvimento de *software*). Esta categoria muitas vezes não é considerada como ferramenta CASE.
- **Modelação de análise e desenho do sistema** – esta é a categoria onde se pode incluir a maioria das ferramentas CASE. Permitem normalmente relacionar modelos de processos com os modelos e requisitos. É nesta área que o paradigma da orientação a objetos e o UML têm tido maior impacto. Como exemplos desta categoria

temos o Rose, o Paradigm Plus, o GDPro. Se considerarmos as ferramentas que adicionalmente suportam abordagens estruturadas, temos o System Architect, o PowerDesigner e o Silverrun.

- **Desenho de bases de dados** – aparecem na sequência das ferramentas anteriores (muitas vezes de forma integrada), mas especializaram-se na definição lógica e física da estrutura das bases de dados. Exemplos: System Architect, PowerDesigner e o Erwin.
- **Programação de aplicações** – ferramentas que normalmente incluem num ambiente único e integrado (se bem que as mais antigas podem ainda funcionam de forma autônoma) funcionalidades de edição de programas, de criação da interface, e outros programas, tais como os interpretadores, compiladores, geradores de código e debuggers. Exemplos: Visual Studio, Eclipse, Delphi, Genexus.
- **Gestão de alterações no software** – dão suporte ao trabalho em equipe, e implementam funcionalidades de gestão de versões, de mecanismos de *check-in* e *check-out* (operações que garantem que apenas uma pessoa acesse com permissões de alteração a um determinado arquivo do sistema), de gestão da configuração e distribuição do software. Exemplos: SVN, CVS, Visual SourceSafe, Rational Team Concert e ClearQuest.
- **Testes**: esta categoria compreende ferramentas que permitem a definição de regras de testes, a geração de scripts para posterior execução de testes, a definição de dados para testes, o controle e a gestão de erros, e a obtenção de estatísticas relacionadas com esta informação. Exemplos: Rational Test Lab Manager, Rational Test Real-Time e TestWorks.
- **Orientadas para a Gestão de Projetos** – são ferramentas cujas principais funcionalidades se destinam a facilitar as tarefas de gestão e coordenação dos projetos, com recursos que auxiliam no planejamento e estimativa de tempos, custos e recursos, utilização de recursos pelo projeto, definição de responsabilidades. Por vezes, incluem ainda facilidades de auxílio na aplicação de uma metodologia de desenvolvimento de software (muitas vezes disponibilizando informação sobre melhores práticas, casos de estudo e uma base de conhecimento sobre todo o processo). Exemplos: MS Project, ProjectBuilder e Juggler.

5.5.5 Vantagens e desvantagens

A introdução de ferramentas CASE numa organização pressupõe uma predisposição para a aplicação de regras e princípios a todo o processo de desenvolvimento, sendo esta precondição um aspecto positivo no processo de melhoria do desenvolvimento de *software* numa organização.

Pressman (2006) identifica algumas das principais vantagens que resultam da aplicação destes tipos de ferramentas.

- Uniformização do processo de desenvolvimento, das atividades realizadas e dos artefatos produzidos.
- Reutilização de vários artefatos ao longo do mesmo projeto, e entre projetos, promovendo o consequente aumento da produtividade.
- Automatização de atividades, com particular destaque ao nível da geração de código e de documentação.
- Diminuição do tempo de desenvolvimento, recorrendo à geração automática de diversos artefatos do projeto, ou à reutilização de outros previamente existentes.
- Integração de artefatos produzidos em diferentes fases do ciclo de desenvolvimento de software, em que as saídas de uma ferramenta são utilizadas como entrada de outra.
- Demonstração da consistência entre os diversos modelos e possibilidade de verificar a correção do software.
- Qualidade do produto final superior, pois a utilização de ferramentas impõe um rigor que obriga a uma abordagem mais estruturada no processo de desenvolvimento.

Porém, existem aspectos negativos e as vantagens citadas podem mesmo ser contrariadas por alguns problemas. No passado, verificou-se frequentemente que os resultados obtidos não estavam de acordo com as elevadas expectativas criadas, o que provocou o fracasso da introdução das ferramentas CASE nas organizações. Um dos fatores mais referidos como responsáveis por esta situação é o elevado tempo de aprendizagem, por vezes requerido para obter o melhor resultado destas ferramentas, e que não é compatível com as exigências de as organizações apresentarem resultados o mais rapidamente possível.

Outros problemas detectados incluem a impossibilidade de, numa abordagem mais estratégica, mapear os processos de negócio (modelados em notações

facilmente compreensíveis pelos usuários) em requisitos de informação. Uma área que ainda hoje constitui um dos principais mitos do *software* é a da geração automática de código: desde há muito tempo tem sido um dos principais objetivos que se procura atingir com as ferramentas CASE, mas onde ainda não foram obtidos resultados completamente satisfatórios.

5.5.6 Funcionalidades das ferramentas CASE

A introdução das ferramentas CASE em uma organização pode ser realizada sob várias estratégias, listadas a seguir:

- ***Suite***: seleção de um conjunto integrado de ferramentas, todas do mesmo fornecedor.
- ***Best-of-breed***: seleção das melhores ferramentas para cada funcionalidade, suportadas por um repositório integrado.
- ***Pontual***: seleção de ferramentas para cobrir áreas pontuais.

Existem algumas funcionalidades que a maioria das ferramentas disponibiliza, independentemente da sua área de intervenção e que estão relacionadas com a gestão e controle do acesso à informação:

- Definição de grupos e de perfis de utilizadores.
- Possibilidade de manter um registro de todas as alterações efetuadas, associado ao controle de versões e à disponibilização de mecanismos de *check-in* e *check-out*.
- Suporte ao trabalho de equipes em um ambiente multiusuário.
- Possibilidade de implementar a noção de projeto e reutilização de artefatos de outros projetos já realizados.

No entanto, é natural que as funcionalidades mais significativas de cada ferramenta sejam especializadas na sua área de aplicação. Por exemplo, uma ferramenta direcionada às atividades relacionadas com gestão de projetos deverá apresentar, entre outras, as seguintes funcionalidades:

- Possibilidade de representar as noções de fase, tarefa e atividade que são executadas ao longo de um projeto.

- Representação de diversos diagramas típicos da gestão de projetos (diagramas de Pert, Gantt, matrizes de alocação de recursos).
- Possibilidade de comparar o esforço planeado com o efetivamente realizado.
- Possibilidade de atribuir atividades a recursos e analisar a alocação de cada um dos recursos.
- Possibilidade do responsável pela execução de uma atividade introduzir informação sobre o trabalho que foi efetuado.
- Possibilidade de utilizar dados históricos para elaborar estimativas para um novo projeto.
- Possibilidade de analisar não apenas as questões de prazos, mas também as financeiras.

Se considerarmos a definição abrangente do conceito de ferramenta CASE, a lista de funcionalidades seria exageradamente grande, o que esta fora do âmbito deste tema. Por isso, vamos apresentar uma lista de funcionalidade dividida em grandes grupos funcionais, de forma a facilitar a compreensão.

- Critérios de modelagem
  - Suporte a um ou mais métodos ou paradigmas metodológicos.
  - Ferramenta orientada para a aplicação de técnicas de modelagem ou de uma metodologia completa. Neste caso, interessa determinar a eventual capacidade de customização da mesma.
  - Tipos de modelagem suportados: modelagem de dados, diagramas funcionais, modelação em UML, modelagem do negócio, modelagem de instalação/distribuição de componentes.
  - Nível de cobertura às várias tarefas do processo de desenvolvimento.
  - Separação e integração entre modelagem de nível conceitual (conceitos do negócio e de análise) e de implementação (desenho e programação).
  - Integração de diversos modelos e capacidade de definir submodelos, possibilitando a existência de diversos níveis de abstração.

- Integração e sincronização entre modelos e interfaces, suportando a devida rastreabilidade.
    - Rastreabilidade dos artefatos ao longo de todo o processo.
    - Possibilidade de converter um modelo entre notações distintas.
    - Possibilidade de estender as representações gráficas, quer em nível dos conceitos, quer do aspecto visual.
    - Automatização de atividades de produção de modelos a partir de outros existentes.
    - Verificação da consistência entre modelos.

- Repositório
    - Tecnologia de implementação do repositório: produto comercial disponível no mercado, base de dados proprietária, ficheiro de texto.
    - Nível de especialização do repositório, podendo suportar apenas uma ferramenta ou possibilitar o acesso por várias ferramentas de modelagem.
    - Estrutura do repositório aberta e conhecida, acessível através de uma interface (SQL, COM, XML).
    - Possibilidade de estender a estrutura do repositório.
    - Facilidade de administração de modo a garantir a segurança do repositório.
    - Suporte a versões.
    - Suporte ao trabalho em equipe.

- Geração de código
    - Possibilidade de análise e verificação da estrutura de um programa.
    - Possibilidade de automatizar a produção do código.
    - Linguagens de programação suportadas para geração de código (por exemplo: C++, Java, Visual Basic).
    - Suporte a engenharia reversa de aplicações existentes.
    - Suporte a *forward-engineering* (geração de código).
    - Suporta desenvolvimento cíclico de código *round-trip engineering*, com introdução de marcas e sem perda de informação.

- Possibilidade de integração com ferramentas de gestão de configurações.
    - Possibilidade de geração de interface gráfica.
    - Geração de documentação de validação do processo.

- Gestão de configuração e das alterações
    - Comparação de alterações entre versões e produção de scripts que reproduzam essas alterações.
    - Produção de relatórios de análise de impacto das alterações a efetuar.
    - Disponibilização de mecanismos de *check-in* e *check-out* de componentes.
    - Definição de níveis de segurança e auditoria aos modelos.

- Modelagem de dados
    - Integração entre os modelos de análise e a estrutura de uma base de dados. Se pensarmos no caso do UML, diagramas de classes.
    - Suporte para Engenharia reversa de bases de dados.
    - Suporte para geração de outros objetos de uma base de dados (por exemplo: *triggers* e *stored-procedures*).
    - Manutenção de integridade referencial.
    - Geração de esquemas de bases de dados, sendo particularmente relevante o suporte para o modelo relacional.
    - Suporte a vários tipos de banco de dados (Oracle, SQL Server, MySQL, SyBase etc.).

- Documentação
    - Geração automática de relatórios e de documentação geral do projetos: estatísticas, dicionário de dados, modelos elaborados, inconsistências, rastreabilidade, operações realizadas.
    - Possibilidade de publicação dos modelos em interface Web.
    - Nível de detalhe que se pode definir para cada atributo/elemento manipulado pela ferramenta.

- Suporta geração automática de documentos com base em templates.
  - Possibilidades de configurar a geração de documentação.

- Extensibilidade
  - Existência de linguagem de *scripting*.
  - Capacidade de integração com outras ferramentas e processos.
  - Possibilidade de estender a ferramenta quanto à documentação gerada.
  - Possibilidade de introduzir na ferramenta novos conceitos.
  - Outras questões genéricas
  - Facilidade de utilização (ajuda, manuais, controle de consistências, relatórios).
  - Funcionalidades de importação e exportação.
  - Definição de regras de validação.
  - Abrangência de conceitos e de componentes suportados, e nível de detalhe de cada um.
  - Obrigatoriedade de utilização e aplicação de padrões.
  - Suporte a modelos de elevada dimensão (por exemplo: o número de processos ou de entidades num diagrama).

ATIVIDADES

01. Faça uma pesquisa na Internet e verifique quais são as ferramentas CASE mais comentadas e suas principais características.

02. Pesquise e liste pelo menos 5 vantagens de usar ferramentas CASE.

03. Cite e explique as 4 fases do processo unificado.

04. É possível comparar o RUP com algum dos modelos de ciclo de vida apresentados no capítulo 4? Qual? Por quê?

05. Um dos principais pilares do RUP é o conceito de *best practices* (melhores práticas), que são regras/práticas que visam reduzir o risco (existente em qualquer projeto de *softwa-*

re) e tornar o desenvolvimento mais eficiente. No RUP, a passagem pelas fases é chamada de ciclo de desenvolvimento, que gera um *software*, adicionalmente dividido em interações e finalizado com ponto de avaliação para analisar os objetivos e os resultados propostos. A iteração, com nove componentes, é estruturada em *workflows* de processo ou fluxos de trabalho realizados pela equipe de projeto. Ao descrever a estrutura dinâmica da empresa, objetivando o entendimento comum entre os envolvidos, sobre quais processos de negócios devem ser contemplados, a equipe de projeto encontra-se no componente de:

a) análise e projeto.
b) implementação.
c) modelagem de negócio.
d) requisitos.
e) teste.

REFLEXÃO

Em qualquer área na qual exista uma atividade de gestão será necessário ter ferramentas para auxiliar o gestor. As ferramentas CASE possuem uma abrangência muito grande e são excelentes para o profissional de TI acompanhar o processo de *software* sob sua responsabilidade.

Por meio de relatórios que elas geram, o gestor consegue medir, avaliar e prever projetos futuros baseados em dados reais de sua equipe, e isso é muito importante e sinal de maturidade da equipe e do processo. E maturidade de processo é algo que o gestor espera, certo?

Aliar ferramentas e processo é outro objetivo dos profissionais de TI. Como foi visto com o processo RUP, isso é possível e viável. O RUP é um método usado por várias empresas e de vários portes e merece uma atenção especial em estudos futuros. Além disso, assim como o PMP, certificação concedida pelo PMI para gerentes de projeto, o RUP também possui certificações e pode ser um diferencial de carreira para os estudantes e profissionais que desejam se destacar no mercado.

LEITURA

Artigo: melhorando o desempenho dos projetos com processos comprovados e adaptáveis. Disponível em: http://migre.me/oJzYU

Excelente artigo que trata sobre o RUP e processos de *software*.

Livro: Introdução ao RUP – Rational Unified Process. Philippe Kruchten. São Paulo: Ciência Moderna, 2003. Este livro traz mais explicações sobre os conceitos apresentados neste capítulo.

Livro: Utilizando UML e Padrões. Craing Larman. Porto Alegre: Bookman, 2007. Este livro é interessante, porque apresenta vários elementos do RUP aplicado a padrões de projeto.

REFERÊNCIAS BIBLIOGRÁFICAS

CANÊZ, A. S.; **Processo Unificado (PU) - Unified Process**. Disponível em: <http://www.adonai.eti.br/wordpress/2011/06/processo-unificado-pu-unified-process/>. Acesso em: 18 dez. 2014.

WAZLAWICK, R. S. Engenharia de Software: Conceitos e Práticas, 1. ed., Elsevier, 2013. 506p.

MACHADO, I. M. R.; PEREIRA, L. M.; **Processo de Desenvolvimento de Software Livre**: Um Estudo de Caso do Projeto EAD Livre. Simpósio Mineiro de Sistemas, 2006. Disponível em: <http://homepages.dcc.ufmg.br/~ivre/Artigo1.pdf>. Acesso em: 17 dez. 2014.

PRESSMAN, R. S. **Engenharia de Software**, 7ª edição. São Paulo: McGraw-Hill, 2011. p. 780.

INFOXZONE. **Práxis - Engenharia de Software**. 2010. Disponível em: <http://infoxzone.wordpress.com/2010/02/10/praxis-engenharia-de-software/>. Acesso em: 15 dez. 2014

MARTINS, V.; **Processo Unificado**. Disponível em: <http://www.batebyte.pr.gov.br/modules/conteudo/conteudo.php?conteudo=1227>. Acesso em 15 dez. 2014.

SOMMERVILLE, I. **Engenharia de Software**, 6. ed. Editora Pearson do Brasil, 2003. p. 292.

SOMMERVILLE, I. **Engenharia de Software**, 7. ed. Editora Pearson do Brasil, 2007.

GABARITO

Capítulo 1

01. Para o desenvolvimento de produtos de *software* de qualidade, é sim necessário o uso de processos de desenvolvimento de *software*, uma vez que a qualidade está diretamente ligada aos processos adotados e à sua prática adequada, desde que o resultado final atinja as necessidades e expectativas do cliente. Caso não fosse adotado, as atividades seriam um caos e muitos fatores de risco estariam presentes no processo de desenvolvimento, não garantindo a qualidade nem a repetibilidade e controle. Sem processos estabelecidos, a possibilidade de gestão fica reduzida.

02. As organizações possuem problemas por uma falta de gestão de processos, que são

características típicas de imaturidade em relação ao nível de seus processos internos e que são bem conhecidos pela indústria.

03.

Sumário

Histórico de Revisão

1. Introdução

1.1. Propósito

1.2. Convenções do documento

1.3. Público-alvo e sugestão de leitura

1.4. Escopo do projeto

1.5. Referências

2. Descrição geral

2.1. Perspectiva do produto

2.2. Características do produto

2.3. Classes de usuários e características

2.4. Ambiente operacional

2.5. Restrições de projeto e implementação

2.6. Documentação para usuários

2.7. Hipóteses e dependências

3. Características do sistema

3.1. Características do sistema 1

3.2. Características do sistema 2

3.3. Características do sistema n

4. Requisitos de interfaces externas

4.1. Interfaces do usuário

4.2. Interfaces de *hardware*

4.3. Interfaces de *software*

4.4. Interfaces de comunicação

5. Outros requisitos não funcionais

5.1. Necessidades de desempenho

5.2. Necessidades de proteção

5.3. Necessidades de segurança

5.4. Atributos de qualidade de *software*

6. Outros requisitos

Apêndice A: Glosário

Apêndice B: Modelos de análise

Apêndice C: Lista de problemas

04. Não. A codificação (implementação) é uma fase do processo de desenvolvimento de *software*. Esta fase, ou etapa, pode ser também chamada de fase de programação ou desenvolvimento. Mas o desenvolvimento de *software* é um conjunto de fases e etapas que completam todo o processo, como concepção, análise e projeto, codificação, testes e manutenção.

05. Os requisitos são essenciais para o desenvolvimento de um *software* ou sistema. Uma das primeiras etapas no processo de desenvolvimento de *software* é fazer o levantamento dos requisitos do sistema e posteriormente efetuar a sua análise. A análise de requisitos possui alguns pontos importantes envolvendo a área de gerenciamento de projetos, pois tem como responsabilidade relacionar as exigências e necessidades do cliente a propor atividades para atingir seus objetivos por meio de soluções que deverão ser desenvolvidas e entregues em um produto final de *software*. Desta forma, a análise de requisitos abrange o entendimento, ou seja, os estudos das necessidades que o usuário necessita e solicita para que seja transformado em soluções. Esta análise é determinante para o sucesso ou fracasso do projeto.

06. Os tipos de requisitos podem ser representados, basicamente, pelos requisitos, do projeto, requisitos do produto, requisitos funcionais e não funcionais.

07. Letra “d”

Capítulo 2

01. Uma documentação é ágil quando possui estas características: Documentos ágeis maximizam o retorno dos clientes; são “magros e econômicos”; satisfazem a um propósito; descrevem informações que têm menor probabilidade de mudar; descrevem “coisas boas de se saber”; têm um cliente específico e facilitam o trabalho desse cliente; são suficientemente precisos, consistentes e detalhados; são suficientemente indexados.

02. Durante o processo de desenvolvimento de *software* pode-se documentar praticamente tudo que necessitar ao longo de todas as etapas do processo, desde anotações, diagramas, fluxogramas, modelagem UML, especificações, levantamentos de requisitos, documentos, guias, manuais, código-fonte e outros artefatos relacionados.

03. Para uma manutenção de qualidade e com a eficiência esperada é praticamente impossível efetuar sem acesso ou consulta à documentação do *software*. No entanto, dependendo da complexidade, e se esta for pequena, já conhecida pelos envolvidos, talvez seja sim possível efetuar manutenções pontuais. Neste caso, estamos contando com a experiência das pessoas, mas para o processo formal de manutenção isto não pode ser considerado.

04. Não. Após a entrega do *software* ao cliente, este entra na fase de manutenção e pode sofrer reparos e ajustes, bem como melhorias e evoluir ao longo do tempo, estendendo ainda mais o seu ciclo de vida. Talvez, o que se termine neste momento é a fase de codificação, em prol da conclusão de um projeto de *software*, no entanto o desenvolvimento de um *software* é continuado com a manutenção.
05. É possível efetuar atividades de suporte e manutenção em partes do *software* que tenham sido concluídas e que estejam devidamente entregues ao cliente, se este *software* possuir uma característica modular. Desta forma, espera-se que uma parte não interfira diretamente em outra parte que, eventualmente, esteja ainda em construção. Portanto, se a entrega final contemplar o uso total do sistema, e se não for possível ter um funcionamento modular, dificilmente será possível efetuar atividades de suporte e manutenção.
06. O ciclo de melhoria contínua de um *software* pode ser utilizado enquanto este estiver sendo útil para o negócio. Não há uma data ou uma contagem específica, e quanto mais for adaptado e melhorado, maior será o seu ciclo de vida, devido às melhorias ali realizadas.

Capítulo 3

01. Os impactos seriam positivos, pois a adoção de uma norma em uma empresa traz maturidade e definição dos processos, direcionando-a em um caminho visando qualidade. Isto também pode ter um outro lado, pois como vários processos serão revistos ou até mesmo exigidos, isto também pode provocar uma grande movimentação dentro da organização, causando resistências em muitas pessoas que ali trabalham, bem como sofrer grandes impactos em relação a eventuais novos processos ou readequação dos mesmos.
02. Os níveis do CMMI podem ser assim relacionados:

Inicial. O processo é caracterizado como sendo imprevisível e ocasionalmente caótico. Poucos processos são definidos e o sucesso depende de esforços individuais e, muitas vezes, heroicos.

Repetível. Processos básicos de gerenciamento de projeto são estabelecidos para controle de custos, prazos e escopo. A disciplina de processo permite repetir sucessos de projetos anteriores em aplicações similares.

Definido. Um processo composto por atividades de gerenciamento e engenharia é documentado, padronizado e integrado em um processo-padrão da organização. Todos os projetos utilizam uma versão aprovada e adaptada do processo organizacional para desenvolvimento e manutenção de produtos e serviços tecnológicos.

Gerenciado. Métricas detalhadas dos processos e dos projetos são coletadas. Tanto os processos como os projetos são quantitativamente compreendidos e controlados.

Otimizado. A melhoria contínua do processo é estabelecida por meio de sua avaliação quantitativa, e da implantação planejada e controlada de tecnologias e ideias inovadoras.

Níveis altos de maturidade ou capacidade são difíceis de serem atingidos, porque envolvem mudança da cultura organizacional. Iniciativas de melhoria de processo podem e devem ser tratadas como se fossem projetos (riscos, dependências, planejamento, acompanhamento, papéis e responsabilidades etc.). Outros fatores a serem considerados são o longo tempo para implantação e o alto investimento.

O MPS.BR está descrito através de documentos em formato de guias:

Guia geral: contém a descrição geral do MPS.BR e detalha o modelo de referência (MR-MPS), seus componentes e as definições comuns necessárias para seu entendimento e aplicação.

Guia de aquisição: descreve um processo de aquisição de *software* e serviços correlatos. É descrito como forma de apoiar as instituições que queiram adquirir produtos de *software* e serviços correlatos apoiando-se no MR-MPS.

Guia de avaliação: descreve o processo e o método de avaliação MA-MPS, os requisitos para avaliadores líder, avaliadores adjuntos e Instituições Avaliadoras (IA).

03.

Nível 1– inexistente

Nível 2

Gestão de requisitos

Planejamento de projeto de *software*

Acompanhamento e supervisão de
projeto de *software*

Gestão de subcontratação de *software*

Gestão de configuração de *software*

Nível 3

Foco no processo da organização

Definição do processo da organização

Programa de treinamento

Engenharia de projeto de *software*

Coordenação intergrupos

Gestão integrada de *software*

Revisão por pares

Nível 4

Gestão da qualidade de *software*

Gestão quantitativa de processo

Nível 5
Gestão de alteração de projeto
Gestão de alteração de tecnologia
Prevenção de defeitos

Capítulo 4

01. Os ciclos de vida de *software*, também chamados de processos de *software*, são um conjunto de atividades que levam à produção de um produto de *software* e seus artefatos. Os ciclos de vida descrevem abstratamente a forma como os *softwares* podem ser desenvolvidos de acordo com o contexto no qual estão inseridos.
02. O manifesto ágil é composto por 12 princípios:
Nossa maior prioridade é satisfazer o cliente através da entrega rápida e contínua de *software* com valor.
Mudanças nos requisitos são bem-vindas, mesmo nas etapas finais do projeto. Processos ágeis usam a mudança como um diferencial competitivo para o cliente.
Entregar *software* frequentemente, com intervalos que variam de duas semanas a dois meses, preferindo o intervalo mais curto.
Administradores (*business people*) e desenvolvedores devem trabalhar juntos diariamente durante o desenvolvimento do projeto.
Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte e confie que eles farão o trabalho.
O meio mais eficiente e efetivo de tratar a comunicação entre/para a equipe de desenvolvimento é a conversa "cara a cara".
Software funcionando é medida primordial de progresso.
Processos ágeis promovem desenvolvimento sustentado. Os financiadores, usuários e desenvolvedores devem ser capazes de manter o ritmo indefinidamente.
Atenção contínua à excelência técnica e bom *design* melhoram a agilidade.
Simplicidade – a arte de maximizar a quantidade de um trabalho não feito – é essencial.
As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizadas.
Em intervalos regulares, a equipe reflete sobre como se tornar mais efetiva e então ajusta seu comportamento de acordo com essa meta.
O modelo cascata assume que os requisitos são inalterados ao longo do desenvolvimento; isto, em boa parte dos casos, não é verdade, uma vez que nem todos os requisitos são completamente definidos na etapa de análise. Um dos riscos desta abordagem em caso de ausência de um processo de gestão do projeto e de controle das alterações bem definido,

podendo passar o tempo num ciclo infinito, sem nunca se atingir o objetivo final, ou seja disponibilizar o sistema a funcionar, devido à Dificuldade em responder a mudanças dos requisitos.

03. Letra “e”

04. Letra “e”

Capítulo 5

01. Alguns exemplos de ferramentas que poderão ser citadas são:

System Architect

Rational Rose

Oracle Designer

GDPro

Power Designer

Silverrun

ERWin

Genexus

02. Algumas das principais vantagens das ferramentas CASE:

Uniformização do processo de desenvolvimento, das atividades realizadas e dos artefatos produzidos.

Reutilização de vários artefatos ao longo do mesmo projeto, e entre projetos, promovendo o consequente aumento da produtividade.

Automatização de atividades, com particular destaque ao nível da geração de código e de documentação.

Diminuição do tempo de desenvolvimento, recorrendo à geração automática de diversos artefatos do projeto, ou à reutilização de outros previamente existentes.

Integração de artefatos produzidos em diferentes fases do ciclo de desenvolvimento de *software*, em que as saídas de uma ferramenta são utilizadas como entrada de outra.

Demonstração da consistência entre os diversos modelos e possibilidade de verificar a correção do *software*.

Qualidade do produto final superior, pois a utlização de ferramentas impõe um rigor que obriga a uma abordagem mais estruturada no processo de desenvolvimento.

03. As fases do Processo Unificado podem ser divididas em:

Concepção (Inception) – entendimento da necessidade e visão do projeto,

Elaboração (Elaboration) – especificação e abordagem dos pontos de maior risco,

Construção (Construction) – desenvolvimento principal do sistema,

Transição (Transition ou Deployment) – ajustes, implantação e transferência de propriedade do sistema

04. Apesar de parecer um modelo em cascata, na verdade cada fase é composta de uma ou mais iterações. Estas iterações são em geral curtas (1-2 semanas) e abordam algumas poucas funções do sistema. Isto reduz o impacto de mudanças, pois quanto menor o tempo, menor a probabilidade de haver uma mudança neste período para as funções em questão.Além das fases e iterações, existem os *workflows*. Cada *workflow* é na verdade uma sequência de tarefas encadeadas e relacionadas a um aspecto importante do projeto, tal como análise do negócio, testes etc.

05. Letra "c"